
BIG
DATA
ALSO
TOOLS
DISK
TARGET
APPLIED
SENSOR
MAY
SHARED
DEFINITION
MOVING
EXAMPLES
EVERY
RECONSIDER
WITHIN
ZETTABYTES
PRACTITIONERS
CAPTURE
THOUGHT
INTERNET
BUSINESS
SETS
GENOMICS
DISTRIBUTED
DESCRIBING
RADIO-FREQUENCY
COMPLEXITY
MANAGEMENT
TERABYTES
CASE
PETABYTES
INCLUDE
ABILITY
ELAPSED
SYSTEMS
TOLERABLE
STORAGE
SIZE
MPP
PARALLEL
SAN
QUALITIES
MASSIVELY
GROW
SINCE
DIFFICULTY
SEARCH
RECORDS
DATABASES
NETWORKS
HUNDREDS
DESKTOP
CAPACITY
NOW
BIOLOGICAL
BURIED
PROCESSING
WORLD'S
UBIQUITOUS
SOLID
TYPES
CURRENTLY
GARTNER
WORKING
OPPORTUNITIES
AMOUNT
TENS
COMPLEX
ANALYTICS
LARGER
CONTINUES
SOCIAL
INDEXING
CITATION
SET
USE
RELATIONAL
ORGANIZATIONS
CONNECTOMICS


GAZİ ÜNİVERSİTESİ
Gazi
1926

# GRAF VERİ TABANLARI ve BÜYÜK ÖLÇEKLİ GRAF VERİ TABANLARINDA ETKİN VERİ İŞLEME YÖNTEMLERİ

**İshak KUTLU**

**DÖNEM PROJESİ**
**BİLGİSAYAR BİLİMLERİ ANA BİLİM DALI**

**GAZİ ÜNİVERSİTESİ**
**BİLİŞİM ENSTİTÜSÜ**

**HAZİRAN 2023**

İshak KUTLU tarafından hazırlanan GRAF VERİ TABANLARI ve BÜYÜK ÖLÇEKLİ GRAF VERİ TABANLARINDA ETKİN VERİ İŞLEME YÖNTEMLERİ" başlıklı Dönem Projesi tarafımdan Bilgisayar Bilimleri Ana Bilim Dalında Dönem Projesi olarak kabul edilmiştir.

**Danışman:** Prof. Dr. M. Ali AKCAYOL

Bilgisayar Mühendisliği Ana Bilim Dalı, Gazi Üniversitesi

Bu çalışmanın, kapsam ve kalite olarak Dönem Projesi olduğunu onaylıyorum. ...……

Dönem Projesi Teslim Tarihi: ...../06/2023

Danışmanı tarafından kabul edilen bu çalışmanın Dönem Projesi olması için gerekli şartları yerine getirdiğini onaylıyorum.

……..……

Prof. Dr. Aslıhan TÜFEKCİ

Bilişim Enstitüsü Müdürü

## ETİK BEYAN

Gazi Üniversitesi Bilişim Enstitüsü Tez Yazım Kurallarına uygun olarak hazırladığım bu tez çalışmasında;

- Tez içinde sunduğum verileri, bilgileri ve dokümanları akademik ve etik kurallar çerçevesinde elde ettiğimi,
- Tüm bilgi, belge, değerlendirme ve sonuçları bilimsel etik ve ahlak kurallarına uygun olarak sunduğumu,
- Tez çalışmasında yararlandığım eserlerin tümüne uygun atıfta bulunarak kaynak gösterdiğimi,
- Kullanılan verilerde herhangi bir değişiklik yapmadığımı,
- Bu tezde sunduğum çalışmanın özgün olduğunu,

bildirir, aksi bir durumda aleyhime doğabilecek tüm hak kayıplarını kabullendiğimi beyan ederim.

İmza
İshak KUTLU
...../06/2023

# *GRAF VERİ TABANLARI ve BÜYÜK ÖLÇEKLİ GRAF VERİ TABANLARINDA ETKİN VERİ İŞLEME YÖNTEMLERİ*

(Dönem Projesi)

İshak KUTLU

GAZİ ÜNİVERSİTESİ

BİLİŞİM ENSTİTÜSÜ

Haziran 2023

ÖZET

Günümüzde büyük veri, önemli bir potansiyel ekonomik değer olarak görülmektedir. Ancak büyük veriden değer yaratma sürecinde geleneksel veri işleme yöntemleri yetersiz kalmaktadır. Literatürde, büyük verinin hacmini işaret eden büyük ölçekli verilerin etkin bir şekilde işlenebilmesi amacıyla graf veri tabanları önerilmektedir. Bu çalışmada graf veri tabanlarının temel yapıları ile çalışma ilkeleri incelenmiş, karmaşık sorguları ve büyük ölçekli verileri etkin bir şekilde işleyebilme yetenekleri araştırılmıştır. Genel bir bakış açısıyla, ölçeklenebilirlik, doğal graf işleme ve indekssiz komşuluk kuralına dayanan doğal graf depolama teknolojileri onların performanslarını açıklayan önemli faktörlerden bazılarıdır. Gelişmiş graf işleme yöntemlerinde anahtar rol oynayan tekrarlı gezinme, patikalar, bilinen yürüyüş, grup etiketleri, denormalizasyon, alt graflar ve performans optimizasyonu gibi graflara ilişkin temel kavramlar ve operasyonlar Gremlin sorgulama dili kullanılarak örneklerle uygulama seviyesinde incelenmiştir. Gelişmiş graf işleme yöntemleri çerçevesinde ise patika bulma ve arama, merkezilik ve topluluk (bölüm ya da küme) algoritmaları odaklı bir çalışma gerçekleştirilmiştir. Bu kapsamda en yaygın kullanılan graf işleme algoritmalarının işleyişi, birbirleriyle ilişkisi, büyük ölçekli graflarda etkinliği araştırılmıştır.

Bilim Kodu : 92414, 92430
Anahtar Kelimeler : Büyük veri analitiği, Graf veri tabanı, Büyük ölçekli veri işleme

Sayfa Adedi : 89
Danışman : Prof. Dr. M. Ali AKCAYOL

# GRAPH DATABASES AND EFFICIENT GRAPH DATA PROCESSING METHODS AT LARGE SCALE GRAPH DATABASES

(Term Project)

İshak KUTLU

GAZİ UNIVERSITY

INSTITUTE OF INFORMATICS

June 2023

## ABSTRACT

Today, big data is seen as a significant potential economic value. However, in the process of creating value from big data, traditional data processing methods are insufficient. In the literature, graph databases are recommended in order to efficiently process large-scale data indicating the volume of big data. In this study, the basic structures and working principles of graph databases are examined, and their ability to efficiently process complex queries and large-scale data is investigated. From a general point of view, scalability, native graph processing and index-free adjacency rule-based native graph storage technologies are some of the important factors explaining their performance. Basic concepts and operations related to graphs such as recursive traversal, pathing, known walking, generic labels, denormalization, subgraphs and performance optimization, which play a key role in advanced graph processing methods, are examined at the application level with examples using Gremlin query language. Within the framework of advanced graph processing methods, a study focused on path finding and searching, centrality and community (partition or cluster) algorithms is performed. In this context, the running of the most widely used graph processing algorithms, their relationship with each other, and their effectiveness in large-scale graphs are investigated.

Science Code : 92414, 92430
Key Words : Big data analytics, Graph database, Large scale processing

Page Number : 89
Supervisor : Prof. Dr. M. Ali AKCAYOL

# TEŞEKKÜR

Bu projenin seçilmesi, araştırılması ve ortaya çıkarılması aşamalarında, ufuk açıcı rehberliği ile yolumu aydınlatan değerli Hocam Prof. Dr. M. Ali Akcayol'a; desteklerini ve anlayışlarını hiçbir zaman esirgemeyen kıymetli eşime ve kızıma sonsuz teşekkürlerimi sunarım.

# İÇİNDEKİLER

**Sayfa**

**Sayfa**

**Sayfa**

# ÇİZELGELERİN LİSTESİ

**Çizelge** **Sayfa**

# ŞEKİLLERİN LİSTESİ

**Şekil** **Sayfa**

**(devam)**

**Şekil** **Sayfa**

# SİMGELER VE KISALTMALAR

Bu çalışmada kullanılmış simgeler ve kısaltmalar, açıklamaları ile birlikte aşağıda sunulmuştur.

| **Simgeler** | **Açıklamalar** |
|---|---|
| **O(.)** | Bir algoritmanın en kötü karmaşıklık zamanı |

| **Kısaltmalar** | **Açıklamalar** |
|---|---|
| **ACID** | Atomize, tutarlılık, bağımsızlık, dayanıklılık |
| **API** | Uygulama programlama arayüzü |
| **BFS** | Genişlik öncelikli arama algoritması |
| **CRUD** | Ekleme, okuma, güncelleme ve silme işlemleri |
| **DFS** | Derinlik öncelikli arama algoritması |
| **ETL** | Çıkart, dönüştür, yükle |
| **IDC** | Uluslararası Veri Kuruluşu |
| **IMDB** | İnternet film veri tabanı |
| **I / O** | Girdi / çıktı |
| **IoT** | Nesnelerin interneti |
| **JAR** | Java arşiv dosyası |
| **LOD** | Bağlantılı açık veri |
| **LPA** | Etiket yayılımı algoritması |
| **LPG** | Etiketli özellik grafı |
| **NoSQL** | İlişkisel veri tabanları dışındaki veri tabanları |
| **OLAP** | Çevrim içi analiz süreci |
| **OLTP** | Çevrim içi işlem süreci |
| **PG** | Özellik grafı |
| **RDF** | Kenar yönlü / etiketli graf |
| **REST** | Temsili durum aktarımı |
| **RID** | Kayıt kimliği |

| | |
|---|---|
| **SCC** | Güçlü bağlanan bileşenler |
| **SOR** | Kayıt sistemi |
| **SQL** | İlişkisel veri tabanı sorgulama dili |
| **WCC** | Zayıf bağlanan bileşenler |

# 1. GİRİŞ

Veri, tek başına herhangi bir değere sahip olmamakla birlikte enformasyona ve bilgiye temel oluşturan ilişkilendirilmeye, gruplandırılmaya, yorumlanmaya, anlamlandırılmaya ve analiz edilmeye ihtiyaç duyan işlenmemiş ham bilgi şeklinde tanımlanabilir (Yılmaz, 2009). Veri tabanı ise verilerin belirli bir amaç doğrultusunda son kullanıcılara sunulması için organize edilerek yeniden yapılandırılması, saklanması, gruplanması, erişime sunulması ve analiz edilmesi gibi veri işleme süreçlerine imkân sağlayan sistemler olarak ifade edilebilir (Doğan ve Arslantekin, 2016).

Büyük verinin nitelikleri çok sayıda olmakla ve sürekli artmakla birlikte, 5V olarak bilinen en yaygın bilinen karakteristikleri hacim (volume), çeşitlilik (variety), hız (velocity), değer (value) ve doğruluk (veracity) şu şekilde özetlenebilir:

- Hacim, verinin tetabayt, perabayt gibi sayısal birim cinsinden miktarını,
- Çeşitlilik, verilerin tablo, ses, görüntü e-posta gibi yapılandırılmış, yarı-yapılandırılmış ya da yapılandırılmamış veri yapılarından oluşabileceğini,
- Hız, verilerin üretilmesi, işlenmesi, saklanması, analiz edilmesi gibi iş süreçleriyle eş zamanlı birleştirilmesini ve / veya makul sürelerde bir değer yaratma amacıyla enformasyona dönüştürülmesini,
- Değer, analiz edilen verilerin organizasyonlar için ekonomik değer üretmesini,
- Doğruluk ise bir dizi işlemden geçen verilerin kalitesini, diğer bir ifadeyle güvenirlik seviyesini ifade eder.

Büyük verinin sadece terabayt ya da petabayt gibi belli bir veri büyüklüğüyle sınırlandırılarak tanımlanamayacağını vurgulayan McKinsey Global Institute (2011) raporuna göre büyük veri, ilişkisel veri tabanları gibi geleneksel veri tabanı yönetim sistemlerinin veri işleme yeteneğinin çok ötesinde bir hacme sahip verilerin yakalanmasını, depolanmasını, yönetilmesini ve analiz edilmesini gerektiren veri setleri olarak tanımlanır (Çelik ve Akdamar, 2018).

Son yıllarda büyük veri akademi, özel sektör, hükümet ve diğer kuruluşların daha fazla ilgilendiği, küresel odaklı bir güç haline gelmiştir. Öyle ki büyük veriye işaret edilerek "veri", 2012 yılında Davos'taki Dünya Ekonomik Forumu'nda para, altın gibi ekonomik

varlıklara benzetilerek “veri”den yeni bir ekonomik değer olarak bahsedilmiştir (Doğan ve Arslantekin, 2016).

Yenilik, rekabet ve üretkenliğin baskın bir kaynağı olarak değerlendirilen büyük verinin ekonomik bir değer olarak kabul edilmesine karşın, ekonomik değerini ortaya çıkarmaya ilişkin süreçler oldukça zordur. Söz gelimi günümüzde pek çok organizasyon kendi ilişkisel veri tabanlarına ek olarak, dış kaynaklardan derlenen veriler üzerinde analiz yaparak da ekonomik değer yaratmak istemektedirler. Ancak ilişkisel veri tabanı gibi geleneksel veri tabanı yönetim sistemleri, dış kaynaklardan derlenen verilerin kurum içi enformasyonun yönetiminde kullanılması konusunda yetersiz kalmaktadır. Çünkü dış kaynaklardan elde edilen veriler xml dosyaları ve meta data gibi yarı yapılandırılmış veya ses, görüntü ve sensör verileri gibi yapılandırılmamış nitelikte olabilir ya da organizasyonların geleneksel veri tabanlarına kolaylıkla aktarılabilir nitelikte olmayabilir (Doğan ve Arslantekin, 2016). Bunun yanında verilerde keşfedilmeyi bekleyen ekonomik değere sahip örüntüleri ilişkisel veri tabanı yönetim sistemleri üzerinde çok sayıda karmaşık sorguyla sürdürülebilir bir şekilde kolaylıkla ortaya çıkarmak mümkün olmayabilir.

Kendisine özgü depolama, veri işleme, analiz etme gibi operasyonları gerçekleştirmeye yarayan veri tabanı yönetim sistemlerine sahip büyük veri teknolojileri, IDC (International Data Corporation) tarafından yüksek hızda yakalama, keşif ve / veya analiz yaparak, çok geniş bir veri çeşidinden ekonomik olarak değer ayıklamak üzere tasarlanmış yeni nesil teknolojiler ve mimariler olarak tanımlamaktadır (Çelik ve Akdamar, 2018). Bu çerçevede sorgulara milisaniyeler içinde tepki veren graf veri tabanları, karmaşık, yarı yapılandırılmış ve yoğun bir şekilde ilişkili büyük ölçekli verilerle başa çıkmak için en iyi çözüm olarak önerilmektedir. Çünkü graf veri tabanlarında verilerin depolandığı düğümler, komşu düğümlerinin doğrudan referansı sahiptir. Bunun anlamı graflarda sorgular, veri tabanının büyüklüğünden bağımsız olarak sadece sorgunun yapılacağı düğüme ilişkin alt grafta gerçekleştirilir (Rawat ve Kashyap, 2017). Diğer bir ifadeyle büyük miktarlarda verilerin depolanabildiği graflarda sorgu süreleri graf veri tabanının boyutundan bağımsızdır.

Graflar nesnelerin interneti (internet of thing, IoT), sosyal ağlar, bilgi (knowledge) grafları, ulaşım ağları, semantik web, bağlantılı açık veri (linked open data, LOD), iletişim, lojistik, veri merkezi yönetimi, biyoinformatik gibi birçok uygulama alanında yaygın olarak kullanılmaktadır. Onlar, esnek modelleme işlevselliği gerektiren büyük veri tabanı

uygulamalarında karmaşık sorguları verimli bir şekilde işleyebildikleri için popülerlik kazanmıştır. Graf ve ilişkisel veri tabanlarını karşılaştıran çalışmalar, grafların çeşitli açılardan ilişkisel veri tabanlarından çok daha iyi performansa sahip olduğunu göstermektedir. Öncelikle avantaj, ilişkisel veri tabanlarında maliyetli olan birleştirme (join) işlemlerini ortadan kaldırmaktır. Graf veri tabanları, düğümleri ve kenarları doğrudan birbirine bağladıkları için birleştirilmeleri gerekmez. Ayrıca graf veri tabanlarında veri ekleme, okuma, güncelleme ve silme (create, read, update, delete, CRUD) operasyonları, ilişkisel veri tabanlarına kıyasla çok daha kolay ve verimli bir şekilde yapılabilir (Ragab, 2020; Asiler, Yazıcı ve George, 2022). Bu sebeple graf veri tabanları, büyük ölçekli veriye ve tekrarlı sorgulara sahip zorlu uygulamalarda diğer veri tabanı modelleri yerine tercih edilir.

Grafların sinirsel bir bilgi ağı oluşturarak birbiriyle kolayca ilişkilendirilebilmesi sebebiyle, geleneksel veri tabanlarının aksine, çok büyük ölçekli ilişkilerin oluşturulmasına olanak sağlar. Graflar bir dizi kural tarafından yönetilebilen veri kümeleri oluşturma yeteneğine sahiptir. Böylece grafın boyutundan bağımsız bir şekilde, veri ilavesi ya da kaldırılması veya verilerin yeniden yapılandırılması söz konusu olduğunda, veri yollarının gerektiği gibi ölçeklendirilmesi ve değiştirilmesi mümkün olur (Lv, Chen, Zheng, Luan ve Guo, 2018). Büyük ölçekli veriler söz konusu olduğunda, grafların ölçekleme kabiliyeti onların en güçlü yanlarından biridir.

Graf veri tabanları, yapısal çerçevede genel olarak iki temel kategoride sınıflandırılabilir. İlki tek bir büyük (global) graftan oluşan veri tabanlarıdır. Bu türden veri tabanlarında sorgulama, yaygın olarak, sorgulanacak ifadeyi temsil eden sorgu grafının tüm örneklerinin global graf içinde aranması yoluyla yerine getirilir. Söz konusu örüntü eşleştirme (pattern matching) literatürde alt graf eşleştirme (subgraph matching) olarak da bilinir ve alt graf izomorfizmi (isomorphism) algoritmalarıyla gerçekleştirilir. Diğeri ise çok sayıda alt (küçük) graftan oluşan veri tabanlarıdır. Moleküler yapılar olarak da adlandırılan söz konusu graflar, düğümler (vertex) ve onları birbirine bağlayan kenarlardan (edge) oluşan alt graflarla temsil edilir. Bu veri tabanları çok büyük boyutlara ulaşabilir. Böyle bir durumda etkin bir sorgulama deneyimi için genellikle alt graflar kullanılır (Luaces, Viqueira, Cotos ve Flores 2021). Bu çalışma kapsamında, global graflardan ziyade, çok sayıda küçük graflardan oluşan büyük ölçekli graf veri tabanları üzerine odaklanılmıştır.

Veri modeline göre en temel iki graf ise RDF (resource description framework) gibi kenar-yönlü / etiketli graf ve özellik grafı (property graph, PG) veri modelidir. Özellik grafı veri modelinde, düğümlere birden fazla etiket eklenebilir, kenarlar ise yönlü ve yönsüz olabilir. Ayrıca grafın düğümleri ve kenarları birden fazla anahtar-değer özellikleri içerebilir. Bu yönleriyle özellik grafları kenar-yönlü / etiketli graflardan daha fazla araç setine sahiptir. Bu sebeple özellik grafları, günümüzde, endüstride ve akademide en yaygın kullanılan ve araştırılan graf veri modelidir (Ragab, 2020). Bu çalışmada da graf veri tabanlarından özellik grafları esas alınmıştır.

Popüler graf veri tabanları arasında Neo4j, Titan, ArangoDB ve HyperGraphDB yer alır. Son zamanlarda, graf veri modellerinin sorgulanması için çeşitli graf sorgulama dilleri önerilmiştir. Özellik grafı modelini destekleyen Gremlin, graf gezinme sorguları için optimize edilmiş işlevsel bir graf sorgulama dili olarak önerilmektedir. Oracle, özellik grafı veri modelini sorgulamasını da destekleyen, SQL benzeri bir graf sorgulama dili olan PGQL'yi tasarlamıştır. Facebook web verilerine erişmek için GraphQ'yu sunmuştur. Neo4j, özellik grafı veri modelini doğal ve sezgisel bir şekilde sorgulamayı hedefleyen ana sorgulama dili olarak Cypher'i tasarlamıştır. Uygulamada, Cypher şu anda en popüler grafik sorgulama dilidir ve SAP HANA, RedisGraph, Agens Graph, MemGraph ve Morpheus (Apache Spark için özel olarak geliştirilen Cypher) dahil olmak üzere diğer birçok graf tabanlı proje ve graf veri tabanı tarafından desteklenmektedir (Ragab, 2020). Bu çalışmada Apache Tinkerpop projesi kapsamında geliştirilen Gremlin sorgulama dili kullanılmıştır.

Graf veri tabanlarında etkin veri işleme yöntemleri 5 temel kategoride incelenebilir.

- Graf algoritmaları (graph algorithms)
- Paralel işleme (parallel processing)
- Dağıtık işleme (distributed processing)
- İndeksleme ve ön belleğe alma (indexing and caching)
- Donanım hızlandırma (hardware acceleration)

Bu çalışma kapsamında graf algoritmaları detaylı incelenecek olup diğer yöntemlerden kısaca bahsedilecektir. Grafları işlemek ve analiz etmek için kullanılan graf algoritmaları, graf veri tabanında belirli bir görevi gerçekleştirmek için tasarlanmış bir dizi prosedürden oluşur. Graflarda gezinme, toplulaştırma ve gruplandırma şeklinde görevleri yerine getiren

sayısız graf algoritması vardır. Graflar, CRUD operasyonları için düğümler ve kenarlar ağını hızla geçebilen graf arama ve gezinme algoritmaları aracılığıyla verilerin verimli bir şekilde sorgulanmasına olanak tanırlar. Bunlardan genişlik öncelikli arama (breadth-first search, BFS) ve derinlik öncelikli arama (depth-first search, DFS) en temel algoritmalardır. Merkezilik algoritmaları, graf düğümleri arasında bir tür "önemlilik" derecelendirmesi fikrine dayanır. Google arama motorunun arkasındaki anahtar algoritmalardan biri olan pagerank algoritması, en önemli merkezilik algoritmalarından biridir. Küme (cluster), bölüm (partition) vb. farklı şekillerde adlandırılabilen topluluk (community) algılama algoritmaları grup içi ilişkilere daha fazla ağırlık vererek graf verilerinin analizinde kullanılır.

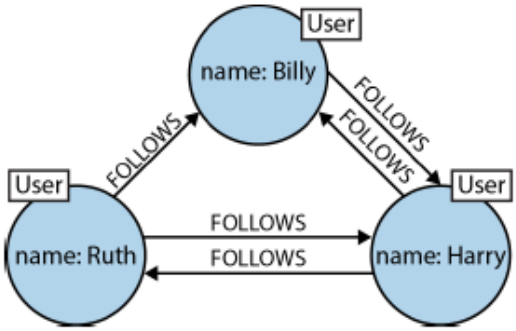
Bu çalışma üç ana başlıktan oluşmaktadır: graf veri tabanları, temel graf işleme teknikleri ve gelişmiş graf işleme yöntemleri. Graf veri tabanları bölümünde graflar ve graf veri tabanları tanıtılmış, onların temel karakteristikleri ve işleyişi incelenmiş ve performans açısından grafların en belirgin özellikleri üzerinde durulmuştur. Temel graf işleme teknikleri bölümünde ise grafların daha yakından analiz edilebilmesi amacıyla, Gremlin sorgulama diliyle graflarda temel işlemler ve etkin veri işleme yöntemleri incelenmiştir. Gelişmiş graf işleme yöntemleri çerçevesinde ise büyük ölçekli graf veri tabanlarında etkin veri işleme yöntemleri özetlenmiştir. Etkin veri işleme yöntemlerinden graf algoritmaları bu bölümün konusunu oluşturmaktadır. Sonuç bölümünde ise graf veri tabanları ve büyük ölçekli graf işleme yöntemleri çerçevesinde, yapılan araştırmanın en önemli bulguları sunulmuştur.

# 2. GRAF VERİ TABANLARI

Verileri önceden tanımlanmış şemalara sahip tablolarda depolayan ilişkisel veri tabanlarının aksine, graf veri tabanları şema içermez. Graflar, ilişkilere dayalı esnek veri modellemeye izin verir. Ayrıca yüksek oranda ölçeklenebilirler ve büyük ölçekli verileri ve karmaşık sorguları işleyebilirler.

## 2.1. Graf Kavramı

Çizge olarak da adlandırılan graf, düğümlerden (nodes, vertices) ve kenarlardan (edges, relationships, arcs) oluşan grafiksel veri yapısıdır. Literatürde düğüm yerine obje / nesne / tepe, kenar yerine ise ilişki / ayrıt / bağlantı ifadeleri de kullanılmaktadır. Graf veri yapısında her düğüm kişi, yer, kategori gibi nesneleri (entities), kenarlar ise bu nesnelerin birbiriyle ilişkilerini temsil eder. Aşağıda yer alan Şekil 2.1'de üç kullanıcılı temsili bir sosyal medya platformu için basit bir graf veri yapısı gösterilmiştir.




Şekil 2.1. Üç düğümlü basit bir graf veri yapısı, (Robinson ve diğerleri, 2015: 2)

Her bir kişi için "User" olarak etiketlenmiş birer düğüm vardır ve her kenar kullanıcıların birbiriyle olan ilişkilerini ifade eder. Billy ve Harry ile Harry ve Ruth birbirini takip etmektedir. Ancak Ruth, Billy'i takip etmesine rağmen, Billy, Ruth'u takip etmemektedir. Ayrıca örnekten açıkça anlaşılacağı üzere kenarların bir yönünün de olabileceği belirtilmelidir. Diğer bir ifadeyle düğümler arasındaki ilişkiyi temsil eden kenarların birer yönü olabilir.

Graflar popüler anlayışın aksine sadece sosyal medya alanında değil, iletişim, lojistik, veri merkezi yönetimi, biyoinformatik gibi çeşitli alanlardan verileri işlemede çok kullanışlı bir veri modelidir. Gerçek dünyada, ilişkisel veri tabanlarının aksine sadece yapılandırılmış

veriler değil, yarı-yapılandırılmış ve yapılandırılmamış çok çeşitli veri türleri ile bunların aralarında ilişkiler vardır. Bu çerçevede graflar, nesneler ve onların arasındaki "ilişkileri" ön plana çıkaran ve tüm veri türleri üzerinde analiz yapmaya olanak tanıyan kapsamlı bir veri modelidir.

Sayısız graf türü olmakla birlikte, temel olarak graflar iki kategoride incelenebilir. Bunlardan birincisi bir uygulamadan gerçek zamanlı veri okuma, yazma gibi çevrim içi işlemleri gerçekleştiren, OLTP (Online Transactional Processing) olarak adlandırılan ve ilişkisel veri tabanlarına kategorisine karşılık gelen "graf veri tabanları"dır (graph databases). Diğeri ise karar destek sistemleri ve raporlama gibi veri analizine ilişkin bir dizi karmaşık görevi yerine getiren ve OLAP (Online Analytical Processing) olarak adlandırılan "graf hesaplama motorları"dır (graph compute engines). Graf hesaplama motorlarından ağırlıklı olarak veri madenciliğinde yararlanılır (Robinson ve diğerleri, 2015: 4). Bu çalışmanın odağı OLTP türü graf veri tabanlarıdır.

### 2.2. Graf Veri Tabanlarının Yükselişi

Graf veri tabanı, temel graf elemanları olan düğümleri ve kenarları bir sorgu diliyle birleştiren, yüksek düzeyde bağlantılı verilerin depolanması ve hızlı bir şekilde çağrılması için optimize edilmiş bir veri depolama motorudur (Bechberger ve Perryman, 2020: 6). Diğer taraftan graf veri tabanı (graph database) kendine özgü ekleme, okuma, güncelleme ve silme (create, read, update ve delete, CRUD) metotlarıyla graf veri modelini ortaya çıkaran online bir veri tabanı yönetim sistemi olarak da ifade edilebilir. Graf veri tabanları genellikle OLTP sistemlerinde kullanılmak üzere tasarlanırlar ve işlem perfomansı için optimize edilirler. Bazı graf veri tabanları ilişkisel veri tabanlarının ACID (atomicity, consistency, isolation, durability) karakteristiklerini de sağlayabilir.

Diğer veri tabanı sistemlerinde veriler arasındaki ilişkiler, ilave anahtarlar kullanmayı gerektiren bir dizi sorgulama tekniğiyle ortaya çıkarılırken, graf veri tabanlarında veriler sahip oldukları "ilişkiler"iyle birlikte depolanırlar. Çünkü graf veri tabanları, nesneler (entities) arasındaki ilişkileri nesnelerin kendisinden daha fazla ön plana çıkaran bir veri depolama yaklaşımını benimser. (Bechberger ve Perryman, 2020: 6). Diğer bir ifadeyle graf veri tabanlarında "ilişkiler" birincil öneme sahiptir. Böylece graf veri tabanının sonuçları,

ilişkisel ve diğer NoSQL veri tabanlarına göre daha basit ve anlamlı olmasının yanında, (özellikle de veri tabanı büyük ölçekli ise) önemli ölçüde hızlıdır.

İlişkisel veri tabanları yüksek derecede bağlantılı verileri işleyebilmesine rağmen sınırlı bir kapasiteye sahiptir. Graf veri tabanları özellikle tekrarlı sorgular (recursive queries), farklı sonuç türleri ve patikalarla ilgili sorguları ilişkisel veri tabanlarından daha etkin bir şekilde işleyebilir. Tekrarlı sorgular, sonlandırma koşulu gerçekleşene kadar kendilerini birçok kez yineleyici bir şekilde çağırırlar. İlişkisel veri tabanları özellikle de sınırsız tekrarlı ve ilişkisel sorguları söz dizimi ve performans açısından etkin bir şekilde işleyemez. Ancak graf veri tabanları bu türden sorguları, zengin ilişki temsillerini kullanarak etkin bir şekilde gerçekleştirebilir (Bechberger ve Perryman, 2020: 9-12). Örneğin "Arkadaşımın arkadaşının arkadaşları kimlerdir?", "Ahmet ve Mehmet nasıl tanıştılar?", "X şirketinin Y şirketiyle ilişkisi nedir?" ya da "X şirketindeki bir yöneticiyle tanışmanın en kolay yolu nedir?" gibi soruların cevapları, ilişkili ya da tekrarlı veriler içerdiğinden graf veri tabanlarından kolaylıkla derlenebilir. Ancak ilişkisel veri tabanlarında bu tür sorgular çok sayıda tabloda "join" operatörlerinin kullanılmasını gerektiğinden verimsiz bir şekilde gerçekleştirilir.

İlişkisel veri tabanları farklı sonuç türlerini döndürmede graf veri tabanları kadar başarılı değildir. Örneğin sipariş ve ürün bilgilerini birlikte döndüren bir SQL sorgusunda, tabloların birleştirilmesinden dolayı "null" değerlere sahip kayıtlar yer alacaktır. Aralıklı veri ya da matris (sparse data ya da sparse matrix) olarak adlandırılan bu durumda, döndürülen veri miktarı artarken verinin yapısal bütünlüğü de azalır. Graf veri tabanlarında ise herhangi bir tablo birleştirme işlemi yapılmadığından dolayı döndürülen sonuçlar arasında "null" değerler bulunmaz (Bechberger ve Perryman, 2020: 12-14). Bu yüzden farklı veri türlerinin bir arada döndürüleceği graf sorgularında aralıklı veri problemiyle karşılaşılmaz ve verinin yapısal bütünlüğü korunur.

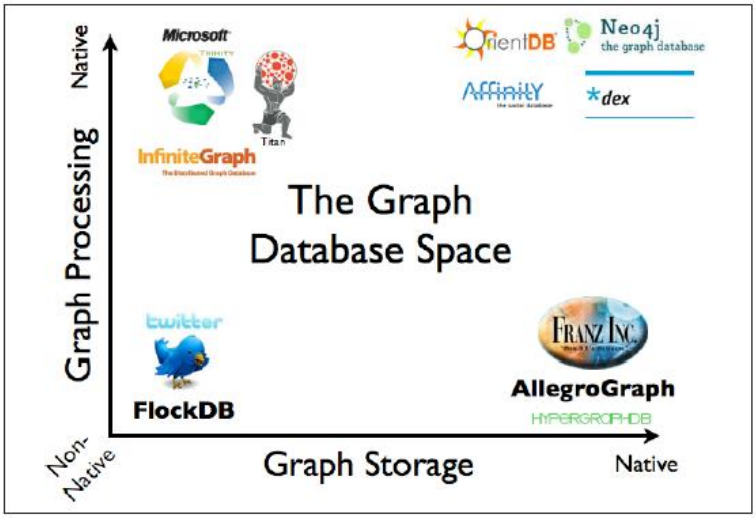
Bir nesnenin diğer bir nesneye göre önemini işaret eden merkezilik (centrality), kümeleme (clustering) ve nüfuz / etki (influence) türünde sorgular, graf veri tabanları tarafından daha verimli işlenir. "LinkedIn'deki bağlantılarım arasında en nüfuzlu / etkili kişi kim?", "Ağımdaki ekipmanlardan hangisinin bozulması en büyük etkiye / zarara yol açar?" ya da "Hangi parçalar aynı anda bozulma eğilimdedir?" soruları, sırasıyla merkezilik, nüfuz / etki ve kümeleme türünde sorgulara örnek verilebilir (Bechberger ve Perryman, 2020, s.18).

## 2.3. Graf Veri Tabanı Sistemleri

Graf veri tabanı sistemleri, genellikle büyük ölçekli ve/veya aşırı derecede ilişkili (karmaşık) verilerle başa çıkmak için tasarlanır. İlişkisel veri tabanlarında dakikalar ya da saatler süren sorgular, graf veri tabanlarında milisaniyeler içinde gerçekleştirilir. Graf veri tabanlarının çerçevesini büyük ölçüde, CRUD (cread, read, update, delete) operasyonlarında kullanılan teknikler ile doğal graf depolama ve doğal graf işleme (indekssiz komşuluk) yapısı belirler. Diğer taraftan graf veri tabanlarında CRUD operasyonları, grafın tamamında değil, kenarlarla birbirine bağlı (sadece) ilgili düğümlerden oluşan daha küçük bir (alt) grafta gerçekleştirilir. Bu yüzden gezinmeler grafın boyutundan bağımsız olarak hızlı bir şekilde yerine getirilir.

Tüm graf veri tabanları doğal graf depolama ve doğal graf işleme teknolojilerini içermeyebilir. Ancak düğümler arasında hızlı gezinmeler için doğal graf depolama ve doğal graf işleme (indekssiz komşuluk) teknolojileri kritik bir öneme sahiptir. Bu teknolojiler herhangi bir düğümün, kendisiyle bağlantılı tüm düğümlerin doğrudan referansına sahip olduğuna işaret eder. Böylece graf üzerinde hızlı bir gezinme mümkün olur. Graf veri tabanlarının sorgulama performansları değerlendirilirken onların söz konusu iki temel özelliği göz önünde bulundurulmalıdır.

- *Graf depolama* (graph storage): Bazı graf veri tabanları grafları yönetmek ve depolamak için tasarlanan "doğal graf depolama"yı (native graph storage) kullanırken, diğerleri ilişkisel ya da nesne-yönelimli veri tabanlarını kullanır. Doğal graf depolama teknolojisine sahip veri tabanları doğal olmayan graf depolama (non-native storage) teknolojilerine göre genellikle daha hızlıdır.
- *Graf işleme motoru* (graph processing engine): Doğal graf işleme (native graph processing) olarak da adlandırılan graf işleme motoru, CRUD operasyonlarını indekssiz komşuluk (index-free adjacency) kuralına göre yapar. Diğer bir ifadeyle veri tabanındaki düğümler fiziksel olarak birbirlerini işaret eder. Bu yüzden indekssiz komşuluk kuralını kullanan graf veri tabanları önemli bir performans avantajına sahiptir.




Şekil 2.2. Graf veri tabanı uzayının genel görünümü, (Robinson ve diğerleri, 2015: 6)

Doğal graf depolama ile doğal graf işleme teknolojilerinden birinin diğerini üstünlüğü olmayıp ikisi arasında bir değiş-tokuş (tradeoff) vardır. Şekil 2.2'de graf depolama ve işleme teknolojilerine göre günümüzde yaygın olarak kullanılan bazı graf veri tabanları gösterilmektedir (Robinson ve diğerleri, 2015: 5).

NoSQL veri tabanlarından olan graflar, genellikle OLTP odaklı olup özellik grafı (property graph) modelini tercih ederler. Bununla birlikte müşteri ihtiyaçları çerçevesinde, günümüzde, farklı graf veri modelleri esas alınarak tasarlanan çok sayıda graf veri tabanı yönetim sistemi de bulunmaktadır. Graf veri tabanlarının çeşitliğini ve çalışma ilkelerini incelemek amacıyla, takip eden alt başlıklarda bazı graf veri tabanlarının en önemli karakteristikleri üzerinde durulmuştur.

### 2.3.1. Neo4j

Lisanslı ürünlerinin yanı sıra açık kaynak sürümleri de bulunan Neo4j, ilk olarak 2007 yılında piyasaya sürülmüştür. Lokal (disk tabanlı) bir graf veri tabanı olan Neo4j, milyarlarca düğüme ve kenara kadar ölçeklenebildiğinden, yüksek derecede bağlantılı ve/veya büyük ölçekli veriler için en yaygın kullanılan graf veri tabanıdır. Neo4j teknolojisinde düğümler ve kenarlar "özellik" (property) içerebilir. Özellik değerleri etiket (label) yapısı içinde saklanır. Bu çerçevede etiketli özellik grafı (labeled property graph, LPG) veri tabanı modelini kullanan Neo4j, kenarlarla ilişkilendirilen düğümlerden oluşur. İndekssiz komşuluk yapısına işaret eden doğal graf işleme ile doğal graf depolama

teknolojilerini kullanır. Neo4j, detayları bu çalışmanın takip eden başlıklarında ele alınacak olan sabit boyutlu kayıtlara dayalı bir depolama tasarımı kullanır. İlişkisel veri tabanlarının en önemli karakteristiklerinden biri olan ACID özelliklerini sağlayan nadir graf veri tabanlarından biridir. Java ve Python gibi yaygın kullanılan çeşitli yazılım dillerini de destekler (Besta ve diğerleri, 2022; Rawat ve Kashyap, 2017).

### 2.3.2. Sparksee / DEX

En yaygın graf veri tabanlarından biri olan Sparksee (eski adıyla DEX), 1 milyona kadar düğümü destekleyebilir. İlk olarak 2008 yılında piyasaya sürülen Sparksee, son derece üretken, kapsamlı ve bitmap tabanlı bir graf veri tabanıdır. Yapılandırılmamış ya da yarı yapılandırılmış verileri ifade eden NoSQL uygulamalarının birçoğunda kullanılabilir. Bir tür sıkıştırılmış zip dosyasına benzetilebilen JAR, resim, ses, video gibi çok sayıda dosyayı içerir ve Java uygulamalarında dağıtık işlemlerde kullanılır. Bu çerçevede DEX, kompakt yapısı sayesinde, yalnızca tek bir JAR (Java archive) dosyasıyla çalıştırılabilir. Java API'leriyle daha verimli çalışan Sparksee, IMDB (internet movie database) gibi uygulamalarda çok başarılı sonuçlar vermektedir. Sparksee (DEX) teknolojisi Java ve .NET yazılım dillerini destekler (Rawat ve Kashyap, 2017).

Sparksee, doğal graf depolama teknolojisini ve LPG modelini kullanan bir graf veri tabanı sistemidir. Her ikisi de nesne olarak adlandırılan düğümler ve kenarlar tekil kimliklere sahiptir. Her özellik isminin, özellik değerlerine göre düğüm ve kenar kimliklerini çözümleyen bir B+ ağacı vardır. Her düğüm için iki bitmap saklanır. Bunlardan biri düğüme gelen kenarları, diğeri ise düğümden giden kenarları gösterir. Ayrıca her kenar yönü için bir B+ ağacı bulunur ve iki B+ ağacı bir kenarın hangi düğümlere bağlı olduğu hakkında bilgi sağlar.

Kayda dayalı olmayan graf veri tabanı sistemlerinden olan Sparksee, kayıtlar yerine B+ ağaçları ve bitmapler olarak uygulanan haritaları (maps) kullanır. Ancak sıkıştırılmamış bitmaplerin boyutu yönetilemeyecek kadar büyüyebilir. Çünkü çoğu graf seyrek (sparse) olduğundan, düğümler veya kenarlar tarafından dizine eklenen bitmapler çoğunlukla sıfır içerir. Bu tür sıfır içeren seyrek bitmaplerin sisteme olan yükünü telafi etmek için bunlar 32 bitlik kümeler halinde kesilir. Bir küme sıfır olmayan bir bit içeriyorsa doğrudan depolanır. Bitmap daha sonra “cluster-id, bit-data” çiftlerinden oluşan bir koleksiyonla temsil edilir.

Bu çiftler, verimli CRUD operasyonlarını tesis etmek amacıyla, sıralanmış bir ağaç yapısında saklanır (Besta ve diğerleri, 2022). Lokal (disk tabanlı) bir graf veri tabanı olan Sparksee teknolojisi Neo4j gibi ACID özelliklerini sağlayan bir graf veri tabanıdır.

### 2.3.3. GBase

LPG tabanlı GBase, yönlü bir graf yapısına sahiptir. Özellikleri ya da etiketleri saklamaz. Seyrek bitişiklik matrisi biçimini (sparse adjacency matrix format) kullanan Gbase teknolojisi, bir grafın bitişiklik (adjacency) matrisinin, engelleyici $O(n^2)$ çalışma zamanı yükünden kaçınmaya odaklanır. Gbase, bir düğüme gelen ve bir düğümden giden kenarların verimli bir şekilde çağrılabilmesi için bitişiklik matrisini sıkıştırmayı amaçlar. Bitişiklik matrisinin eş zamanlı olarak kullanılması, iki rassal düğümün birbirine bağlı olup olmadığının 0(1) çalışma zamanı karmaşıklığıyla doğrulanmasını sağlar. Bitişik matrisinin her satır ve sütununda K adet blok olduğu varsayımında, GBase, onu sıkıştırmak için $K^2$ seviyesinde bloklara ayırır. K parametresi, belirli veri tabanları için optimize edilebilir. Bir bloğun bir dosyada saklandığı varsayımında, K küçüldüğünde, daha fazla küçük dosya; K büyüdüğünde ise daha az büyük dosya çağrılmış olur. Bloklar sadece sıfırlar ve birler içerdiğinden daha yüksek sıkıştırma oranları için daha fazla optimizasyon yapılabilir (Besta ve diğerleri, 2022).

### 2.3.4. OrientDB

Doküman tabanlı bir graf veri tabanı olan OrientDB'de bir doküman d harfi ile temsil edilir. OrientDB'de her dokümanın, (d'nin) depolandığı doküman koleksiyonundan ve onun içindeki pozisyonlardan oluşan bir kayıt kimliği (record ID, RID) vardır. Dokümanlar arasındaki işaretçiler (ya da bağlantılar) için tekil RID'ler kullanılır. OrientDB normal (regular) kenarlar ve hafif (lightweight) kenarlar adı verilen yapıları kullanır. Normal kenarlar, bir kenar dokümanında depolanır ve kenar özelliklerini ya da etiketleri çözümlemek için kendine ait ilişkili anahtar-değer çiftlerine sahip olabilir. Diğer taraftan hafif kenarlar, doğrudan (kaynak ya da hedef) komşu düğümün dokümanında saklanır. Bu tür kenarların ilişkili herhangi bir anahtar-değer çifti bulunmaz. Bunların, diğer düğümler için basit birer işaretçi görevi vardır ve doküman RID'leri olarak uygulanırlar. Bu nedenle, bir düğüm dokümanı sadece düğümün etiketlerini ve özelliklerini değil, aynı zamanda hafif kenarların bir listesini ve komşu normal kenarları gösteren işaretçi (RID) listesini de depolar.

Her normal kenar, kaynak ve hedef düğümü depolayan dokümanlara ilişkin işaretçilere (RID'ler) sahiptir. Her düğüm ise kendisine gelen ve kendisinden giden kenarlara ilişkin bir bağlantı listesi (RID'ler) depolar (Besta ve diğerleri, 2022).

### 2.3.5. ArangoDB

Doküman tabanlı diğer bir graf veri tabanı ArangoDB dokümanları, JSON belgelerinin sıkıştırılmış bir hali olan "VelocyPack" adlı ikili (binary) bir formatta tutar. Dokümanlar farklı koleksiyonlarda saklanabilir ve belirli bir koleksiyon içinde tekil bir kimlik olan _key özelliğine sahip olabilir. OrientDB'den farklı olarak, bu kimlikler doğrudan bellek işaretçileri değildir. ArangoDB, grafları saklamak için düğüm koleksiyonlarını ve kenar koleksiyonlarını kullanır. İlki, düğüm (vertex) dokümanlarını içeren normal (regular) doküman koleksiyonlarıdır. Düğüm dokümanları, komşu kenarlar hakkında hiçbir bilgi saklamaz. Böylece kenarlar eklendiğinde veya çıkarıldığında düğüm dokümanının değiştirilmesi gerekmez. İkincisi ise kenar dokümanlarını depolayan kenar (edge) koleksiyonlarıdır. Kenar dokümanlarının "_from" ve "_to" şeklinde iki özel özelliği vardır. Bunlar belirli bir kenarla birbirine bağlı iki düğümle ilişkili dokümanların kimlikleridir. ArangoDB düğüm dokümanlarının okunmasını engellemek suretiyle bir optimizasyon yapar. Bir kenar dokümanına doğrudan erişim sağlamak için başka bir kenar dokümanındaki düğüm kimliğini kullanır. Bu optimizasyon önbellek verimliliğini arttırarak sorgu süresini azaltabilir (Besta ve diğerleri, 2022). Buradaki amaç, kenarlar üzerinde gezinme (traversal) maliyetinin düğümler üzerinde gezinme maliyetinden düşük olmasının avantajını kullanmaktır.

### 2.3.6. HyperGraphDB

HyperGraphDB, anahtar-değer (key-value) tabanlı, hiper grafları destekleyen açık kaynaklı bir graf veri tabanıdır. Hiper graf, düğüm benzeri yapılara benzeyen hiper kenarlar (hyperedge) yardımıyla ikiden fazla düğümü birbirine bağlayan bir graf yapısıdır. Hiper graflar farklı alanlarda, diyagram bilgilerinin gösterilmesi amacıyla kullanılır. Bunlar bir Java API'si ile birlikte internet sorgularının temelini oluşturur. Düğümleri ve kenarları atom olarak adlandırılan HyperGraphDB'ın atomları, tekil ve kriptografik olarak güçlü bir kimliğe sahiptir. Böylece ilişkileri/kenarları daha fazla indirgemek suretiyle, aynı kimliğe sahip atomlarda tutulan farklı graf öğelerinin çarpışmasını (çakışmasını) önlemek mümkün

olur. Bir hiper kenarın atomu, bu hiper kenarla bağlanan düğümlere karşılık gelen kimliklerin bir listesini saklar. Ayrıca düğümler ve hiper kenarlarda, bir değerin işaret ettiği tekrarlı yapılarda yer alan özellikler gibi ilave veriler depolanır.

Hiper graflara dayanan HyperGraphDB modeli, son derece öngörülemez ve kapsamlı bir öğrenme uygulamasına sahiptir. Dağıtık ve diyagram olarak düzenlenen HypergraphDB veri tabanı, diğer diyagram veri tabanlarından daha kapsamlı olması sebebiyle, çok sayıda uygulama alanına sahiptir. HypergraphDB, hesaplamaya dayalı bilimlerde yaygın şekilde kullanılır. Ayrıca biyoinformatik alanında da sahte bilincin (counterfeit consciousness) bir parçası olarak kullanılmaktadır (Besta ve diğerleri, 2022; Rawat ve Kashyap, 2017).

### 2.3.7. Trinity

Trinity, bir bellek bulutu üzerinden iletilen bir graf yapısına dayanır. Bellek bulutu, bir makine grubu üzerinde depolamaya değer global olarak adreslenebilir belleklerin tümüdür. Bu çerçevede devasa bir diyagram hazırlama makinesine benzetilebilen Trinity, büyük veri kümelerinde hızlı diyagram araştırması ve paralel işleme sağlar. Diğer bir ifadeyle büyük ölçekli veri setlerinin hızlı bir şekilde işlenmesine olanak tanır. C# API desteğine sahip Trinity, milyarlarca dağıtıcıya sahip geniş diyagramlarda yüksek verim sağlar (Rawat ve Kashyap, 2017).

Microsoft'un graf motorunda konumlanan Trinity, anahtar-değer (key-value) depolama yöntemini kullanır. Anahtarlara hücre kimlikleri, değerlere ise hücreler denir. Bir hücre, diğer hücrelerin kimlikleri de dahil olmak üzere farklı veri türlerine ilişkin veri öğelerini tutabilir. Microsoft graf motoru (graph engine), Trinity (anahtar-değer) depolama katmanının üstünde bir graf depolama katmanı sunar. Düğümler, özel bir alanın bir düğüm kimliği veya bu kimliğin bir karmasını içerdiği hücrelerde depolanır. Belirli bir v düğümüne doğrudan komşu kenarların listesi doğrudan v düğümünün hücresinde saklanır. Bununla birlikte zengin veriler içeren bir kenar, ilişkili verilerle birlikte farklı bir hücrede de depolanabilir (Besta ve diğerleri, 2022).

### 2.3.8. Titan

Java'da yazılan Titan, açık kaynaklı bir girişim olarak 2012 yılında piyasaya sürülmüştür. Titan graf veri tabanının en önemli özelliği ölçeklendirme kabiliyetidir. Hadoop sistemi ile grafları işleyen Titan, karmaşık sorgulara milisaniyeler içinde tepki verir. Özellik grafı (property graph) veri tabanı modelini destekler. Doğal graf işleme teknolojisine, rexster sunucuya ve Apache TinkerPop projesi kapsamında geliştirilen Gremlin sorgulama dili desteğine sahiptir. Java API'leri ve Gremlin sorgulama dili kullanılarak diğer uygulamalarla bağlantısı sağlanabilir (Rawat ve Kashyap, 2017).

Titan (ve devamı olan JanusGraph), geniş sütunlu depolar (wide-column stores) üzerine inşa edilmiştir. Apache Cassandra örneğinde olduğu gibi farklı geniş sütun depolarını arka uç (backend) olarak kullanabilirler. Titan'da düğümler satırlarda depolanır. Her düğüm özelliği ve komşu kenar ayrı bir hücrede saklanır. Böylece herhangi bir kenarın tüm özellikleri tek bir hücreden erişilebilir. Hücreler üzerinde verimli aramalar yapmak amacıyla satırlarda yer alan hücreler, anahtarlarına göre sıralanır. Özelliklerin ve kenarların hücre anahtarları, özellik anahtarları önce gelecek şekilde sıralanır. Anahtarlara göre sıralanan tablolar doğrudan tablet" (tablets) adı verilen bölümlere ayrılır. Tabletler birden fazla veri sunucusu üzerinde dağıtık bir şekilde veri işlemeye olanak tanıyan bölümlenmiş tablolardır (Besta ve diğerleri, 2022).

### 2.3.9. WhiteDB

Demet (tuple) tabanlı WhiteDB, demet öğelerinin sayısını ifade eden rassal bir demet uzunluğuyla yeni kayıtların (demetlerin) oluşturulmasını sağlayan bir graf veri tabanı sistemidir. Küçük değerler ve diğer demetleri gösteren işaretçiler (pointers) doğrudan belirli bir alanda saklanır. Büyük dizeler (strings) ise ayrı bir depoda tutulur. Her büyük değer yalnızca bir kez depolanır. Bir referans sayacı, herhangi bir zamanda söz konusu büyük değere başvuran demet sayısını takip eder. WhiteDB sadece tek demet kaydına erişmeyi sağlar. Örneğin belirli bir düğümün tüm komşularını çağıracak bir sorgu motoru veya graf API'si yoktur. Bununla birlikte demetler, düğüm ve kenarların depolanması amacıyla da kullanılır ve bunları bellek işaretçileri aracılığıyla birbirine bağlar. Bu yapı sayesinde WhiteDB, yapısal olarak düzensiz bir grafın örüntülerini hızlıca çıkartabilen çeşitli sorgulara olanak tanır. Örneğin bir v düğümü, özellikleriyle birlikte v ile ilişkili bir demette ardışık

alanlar olarak saklanabilir ve v'nin demetinde v'nin seçilen komşularına ilişkin işaretçiler tutulabilir (Besta ve diğerleri, 2022).

### 2.3.10. Infinite Graph

Infinite Graph, "Objectivity" isimli bir topluluk (association) tarafından geliştirilmiştir. Topluluk, büyük ölçekli veri tabanları üzerine çalışan bir organizasyondur. Infinite Graph, Java tabanlı ve bulut destekli dağıtık bir graf veri tabanıdır. Çok sayıda makineye dağılmış graf veri tabanı, uygulamalardan gelen kilit (lock) taleplerini işleyen bir kilit sunucusuna sahiptir. Büyük ölçekli verileri işlemek için tasarlanmıştır. Farklı seviyede keşifler gerektiren karmaşık ilişkileri yönetme becerisine sahiptir. Düğümlerin ve kenarların yapısı API'lerin kullanımıyla hızlı gezinmeleri garanti eder. Hızlı gezinme sorgularını yerine getiren bu yapı "uyarlanabilir graf algılama cihazı" (adaptable chart perception device) olarak adlandırılır. Bir uygulamadan veri tabanına akıllı erişim için REST arayüzüne ihtiyaç duyulur (Rawat ve Kashyap, 2017).

### 2.3.11. Graf veri tabanı sistemlerinin değerlendirmesi

Graf veri tabanlarında genellikle veri modelleri, programlama dili ve API desteği, sorgulama dili, dağıtıklık ve ACID desteği çeşitlilik göstermektedir. Bu çeşitlilik, her bir graf veri tabanı uygulamasının odaklandığı problem türünden ve/veya çözüme ilişkin yaklaşımından kaynaklanmaktadır. Örneğin okuma yoğun işlemlerde Sparksee (DEX) gibi dağıtık mimariye sahip graf veri tabanları daha yüksek bir verimlilik sunarken, yazma yoğun işlemlerde ise nispeten düşük bir performans gösterebilir. Neo4j gibi bazı veri tabanları ise okuma ve yazma işlemleri arasında bir optimizasyonu tercih eder. Böylece CRUD stratejileri üzerine yoğunlaşarak gezinme hızını iyileştirmeyi amaçlar.

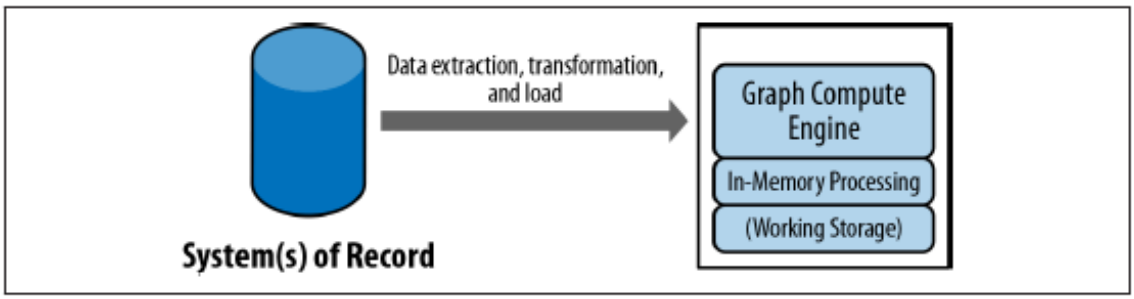
Neo4j, DEX, Titan (BerkeleyDB ve Cassandra) ve OrientDB olmak üzere dört veri tabanının farklı sevide iş yükleri ve parametrelerle karşılaştırıldığı bir araştırmada, en iyi gezinme performansını Neo4j veri tabanı göstermiştir. Okuma yoğun işlemlerde ise Neo4j, Titan-BerkeleyDB ve OrientDB graf veri tabanları, Sparksee (DEX) ve Titan-Cassandra'nın oldukça gerisinde kalmıştır (Rawat ve Kashyap, 2017). Neo4j veri tabanının en iyi gezinme performansına sahip graf olarak diğerlerinden ayrılması, onun gezinme performansına odaklanarak okuma ve yazma işlemlerini dengelemesinden kaynaklandığı söylenebilir.

Ancak okuma yoğun işlemlerde ise Sparksee (DEX) ve Titan-Cassandra'nın Neo4j'yi geride bırakması, onların dağıtık bir mimariye sahip olmasına ve grafların doğasında bir optimizasyon probleminin varlığına işaret eder. Diğer bir ifadeyle grafların tasarımında, okuma, yazma, gezinme, ölçekleme gibi çeşitli performans kriterleri arasında bir tür değiş-tokuş (trade off) gözetilmek zorundadır. Tüm kriterlerle en yüksek performansa sahip olmak grafların dünyasında pek mümkün görünmüyor.

### 2.4. Graf Hesaplama Motorları

Graf hesaplama motoru (graph compute engine), büyük veri setlerini analiz edebilen global hesaplama algoritmaları kullanan bir teknolojidir. Graf hesaplama motorları verilerdeki örüntüleri veya eğilimleri ortaya çıkarma gibi görevleri yerine getirmek için tasarlanırlar.

Graf hesaplama motorları global sorgulara odaklandığından, yığınlar halindeki büyük miktarlardaki veriyi işlemek ve taramak için optimize edilirler. Bu açıdan onlar, veri madenciliği ve OLAP gibi yığın analizi teknolojilerine benzetilebilir. Bazı graf işleme motorları bir graf depolama katmanına sahip olabilirken, bazıları ki muhtemelen bir çoğu sıkı sıkıya, dış kaynaklardan derlenen verileri işlemek ve ardından sonuçları başka bir yerde depolar.




Şekil 2.3. Tipik bir graf hesaplama motorunun genel görünümü, (Robinson ve diğerleri, 2015: 7)

Şekil 2.3'te yaygın bir mimariye sahip graf işleme motoru gösterilmektedir. Mimari MySQL, Oracle ve Neo4j gibi çalışma zamanında bir uygulamadan gelen sorgulara yanıt veren OLTP özelliklerinde bir veri tabanı kayıt sistemi (system of record, SOR) içerir. ETL (extract, transform, load) işlemi, verileri, veri tabanı kayıt sisteminden çevrim dışı sorgulama ve analiz için graf hesaplama motoruna periyodik olarak gönderir (Robinson ve diğerleri, 2015: 7).

## 2.5. Graf Veri Tabanlarının Gücü

Graf veri tabanlarının gücü onların veri işleme performansı, değişen iş koşullarına uyarlanabilme esnekliği ve yazılım geliştirme araçlarıyla uyumluluğu çerçevesinde üç kategoride değerlendirilebilir.

### 2.5.1. Performans

İlişkisel ve diğer NoSQL veri tabanlarıyla karşılaştırıldığında graf veri tabanlarının sunduğu en önemli olanaklardan biri, verilerdeki ilişkileri ortaya çıkarmada gösterdiği kayda değer performanstır. Veri setinin büyümesine paralel, "join" sorgularının önemli bir performans kaybına yol açtığı bilinen ilişkisel veri tabanlarının aksine, graf veri tabanlarında performans sabit kalma eğilimindedir. Çünkü sorgular grafın bir parçasında zaten konumlandırılmıştır. Bu yüzden bir sorgunun çalışma zamanı, tüm grafın büyüklüğüyle değil, sorguyu gerçekleştirmek amacıyla gezinilen graf parçasının (alt grafın) büyüklüğüyle orantılı bir şekilde değişir (Rawat ve Kashyap, 2017). Daha açık bir ifadeyle herhangi bir sorguda gezinme (traversal), graf veri tabanındaki tüm düğümler üzerinde değil, sadece ilgili oldukları düğümler üzerinde yapılır.

### 2.5.2. Esneklik

Graf veri modeli iş gereksinimleri doğrultusunda hızlı aksiyon almaya olanak tanır. Graf veri tabanları, mevcut yapıdaki sorguları ve işlevleri bozmaksızın yeni tür ilişkileri (kenarları), düğümleri, etiketleri ve yeni alt grafları (subgraph) kolaylıkla eklemeye olanak tanır. Graf modellerinin esnekliğinden (flexibility) dolayı, veri tabanına ilişkin alan (domain) değişikliğinde, veri tabanının yeni alana göre yeniden yapılandırılmasına gerek yoktur. Doğası gereği eklemelere eğilimli olan graf veri tabanları, bakım yüklerini ve risklerini düşürmenin yanı sıra, daha az taşınma (migration) gereksinimine ihtiyaç duyar (Robinson ve diğerleri, 2015: 9).

### 2.5.3. Uyumluluk

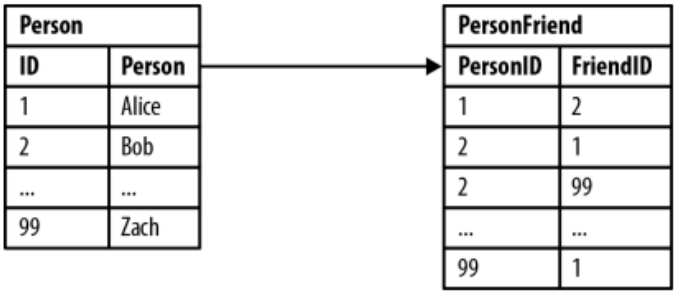
Graf veri modeli, günümüzün sürekli artan yazılım sağlama araçlarıyla uyumlu bir teknoloji kullanır. Özellikle graf veri modelinin şemadan bağımsız doğası, graf veri tabanının test edilebilir API'si ve sorgulama diliyle birleştiğinde kontrollü olarak bir uygulama

geliştirilebilmesine olanak sağlar. Graf veri tabanı geliştirme, günümüzün uyumluluk (agility) ve test odaklı yazılım geliştirme pratikleriyle, ilişkisel veri tabanlarından daha iyi uyum sağlar ve graf veri tabanı destekli uygulamaların değişen iş ortamlarına ayak uydurarak gelişmesine olanak tanır (Robinson ve diğerleri, 2015: 9).

## 2.6. Graf Veri Tabanlarının Etkinliği

Tablo yapısına sahip ilişkisel veri tabanlarına geçici olarak yeni ilişkiler eklenmek veya düzenlenmek istendiğinde, onlar bu ilişkileri yönetmede yeterli etkinliği sunamamaktadır. Esnek olmayan şematik yapılar kullanan ilişkisel veri tabanları, bir veri setinde karmaşık ilişkileri ortaya çıkarmak için yabancı anahtarlara (foreign keys) gereksinim duyar. Yabancı anahtarlar, çeşitlilikler arasındaki ilişkileri ortaya çıkarmak için tabloları "join" operatörü ile birbirine bağlamada kullanılır . Verilerin boyutu, karmaşıklığı ve çeşitliliği arttıkça, "join" sorgularından dolayı ilişkisel veri tabanlarının performansı ciddi ölçüde düşer.

Diğer taraftan ilişkisel veri tabanlarında çift yönlü sorgular oldukça maliyetlidir. "Bir müşteri hangi ürünleri satın aldı?" sorgusuna kıyasla, "Hangi müşteriler bu ürünü satın aldı?" sorgusu daha maliyetlidir. Ancak indekslerin de kullanılmasını gerektiren "Bu ürünü satın alan müşterilerin hangileri ayrıca şu ürünü satın aldı?" türde tekrarlı bir çift yönlü sorguyu kolaylıkla gerçekleştirmek mümkün olmaz. Çünkü ilişkisel veri tabanlarında sorguların yineleme derecesiyle (ya da bağlantılı alanların çeşitliliğiyle) performansı arasında ters yönlü bir ilişki vardır.




Şekil 2.4. İlişkisel veri tabanlarında arkadaşlık ilişkilerinin modellenmesi, (Robinson ve diğerleri, 2015: 13)

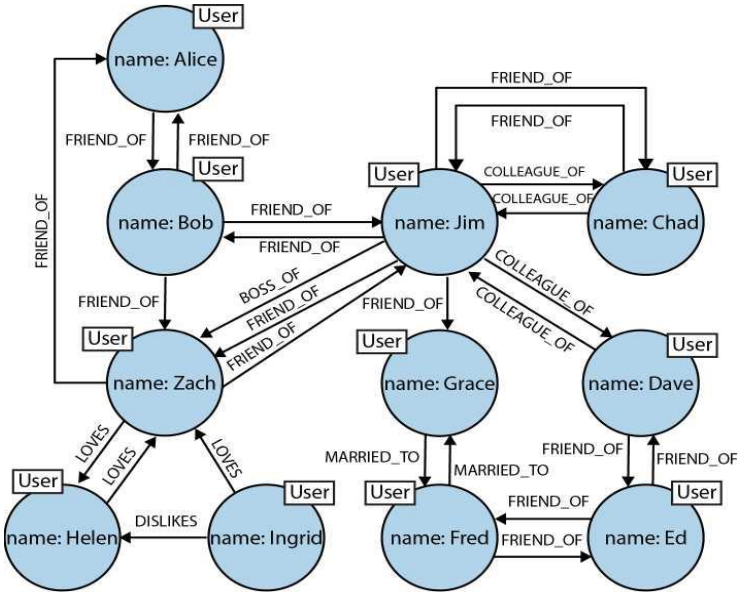
Şekil 2.4'e göre "Bob'un arkadaşları kim?" sorgusu, PersonFriend tablosunun sadece "Bob" (yani PersonID'si 2) ile filtrelenen satırlarını dikkate alacağından kolaylıkla Alice ve Zach değerlerini döndürecektir. Ancak arkadaşlık durumu her zaman simetrik bir özelliğe sahip

olmayabilir. Bu çerçevede “Bob ile arkadaş olanlar kim?” çift yönlü sorgusu, Alice değerini döndürecektir. Ancak bu sorgu PersonFriend tablosunun tüm satırları üzerinde kaba arama (brute force) şeklinde gerçekleşeceğinden bir önceki sorguya göre daha maliyetli olacaktır.

“Bob’un arkadaşlarının arkadaşları kim?” yineleyici sorgusu bir indeksleme gerektirecektir. Böylece Bob’un ikinci dereceden arkadaşları (yani Alice’in arkadaşları), yineleyici “join” sorguları gerektirecek olup söz dizimsel ve hesaplama açısından karmaşık bir sorguyla elde edilebilir. “Bob’un arkadaşlarının arkadaşlarının arkadaşlarının arkadaşları kim?” şeklinde arkadaşlığın derecesini (üçüncü, dördüncü vb.) arttıran bir sorguda ise yineleyici “join” tablolarının hesaplama ve alan karmaşıklığı nedeniyle algoritmanın çalışma zamanı hızla kötüleşir (Robinson ve diğerleri, 2015: 11-13).

Anahtar-değer (key-value), doküman ve kolon (column) tabanlı diğer NoSQL veri tabanları verileri bağlantısız şekilde saklarlar. Bu yüzden veri tabanında saklanan veriler yabancı anahtarlar (foreign keys) kullanılarak birbiriyle ilişkilendirilir. Bu yöntem veri kümelerinin uygulama düzeyinde birleştirilmesini gerektirdiğinden performansı kısıtlayıcıdır. Yabancı anahtarların yalnızca tek bir yönde çalışması, çift yönlü sorgulamaları çok zaman alıcı hale getirirler. Geliştiriciler bu sorunu çözmek için genellikle geriye dönük ilişkiler ekler veya veri kümesini Hadoop gibi harici bir veri işleme motoruna aktarır. Her iki durumda da sonuçlar yavaştır ve ilişkiler yeterince açık bir şekilde elde edilemez (Robinson ve diğerleri, 2015: 16).

İlişkisel ve NoSQL veri tabanlarının aksine, graf veri tabanında ilişkili veriler, Şekil 2.5’te temsil edildiği gibi ilişkileriyle birlikte depolanır.




Şekil 2.5. Bir grafta çok boyutlu ilişkilerin modellenmesi,
(Robinson ve diğerleri, 2015: 19)

Bir graf modelinde nesneler (entities) arasındaki ilişkiler, gerçek dünyadaki ilişkili verilerde olduğu gibi çeşitlilik gösterir. Graf modeli, orijinal verilerde ve onların amacında herhangi bir bozulmaya yol açmadan veya verileri taşımadan, yeni düğümler ve yeni ilişkiler eklenmesine olanak tanıdığı için son derece esnektir. Ağın zengin bir resmini sunan graf modelinde, örneğin kimin kimi "sevdiğini", kim kimin "meslektaşı" ya da "patronu" olduğu gibi ilişkiler açıkça görülebilir.

Graf veri tabanında düğümler oynadıkları rollere göre farklı etiketler (labels) atanarak sınıflandırılabilir. Örneğin bazı düğümler kullanıcıları temsil ederken, diğerleri siparişleri veya ürünleri temsil edebilir. Böylece etiketler, düğümleri gruplandırmak için kullanılabilir. Örneğin veri tabanından "Kullanıcı" etiketli tüm düğümleri bulması istenebilir. Bir düğüm, bir grafta birkaç farklı rolü yerine getirebileceğinden, birden fazla etikete de sahip olabilir (Robinson ve diğerleri, 2015: 18-20).

Graf veri tabanlarının zengin ilişki temsillerinden bir diğeri ise patikalardır. Bir patika (path), graf boyunca gezinmenin gerçekleştiği düğümler (vertex, node) ve kenarlar (edge, relation, connection) dizisidir. Veri tabanı içinden iki nesnenin birbirine bağlanma şeklini döndürme yeteneği, graf veri tabanlarına özgü bir özelliktir (Bechberger ve Perryman, 2020, s.14). Grafın sorgulanması veya grafta gezinme birbirini takip eden patikalar üzerinde gerçekleşir. Dolayısıyla patikaya dayalı graf veri tabanı işlemlerinin verimliliği, diğer bir ifadeyle CRUD

işlemlerinin etkinliği, verilerin veri tabanında düzenlenme biçimiyle (yani veri modeliyle) çok yakından ilgilidir.

Çizelge 2.1'de bir sosyal ağda maksimum beş derinliğe kadar arkadaşların arkadaşlarının bulunmaya çalışıldığı araştırmanın sonuçları yer almaktadır. Her birinin yaklaşık 50 arkadaşı olan 1 000 000 kişiden oluşan bir sosyal ağ için sonuçlar, bağlantılı veriler için graf veri tabanlarının tartışmasız en iyi seçenek olduğunu göstermektedir (Robinson ve diğerleri, 2015: 20).

Çizelge 2.1. Yüksek dereceli sorgulara ilişkisel ve graf veri tabanının tepki süreleri, (Robinson ve diğerleri, 2015: 21)

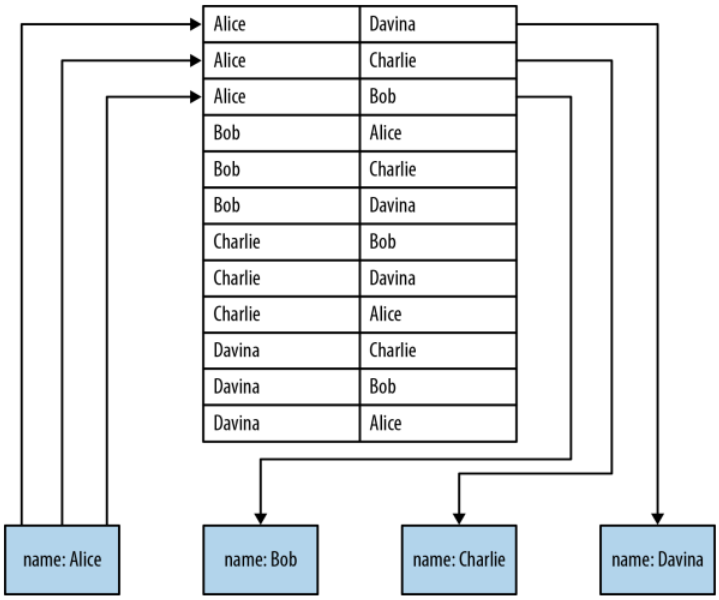
| Depth | RDBMS execution time(s) | Neo4j execution time(s) | Records returned |
|---|---|---|---|
| 2 | 0.016 | 0.01 | ~2500 |
| 3 | 30.267 | 0.168 | ~110,000 |
| 4 | 1543.505 | 1.359 | ~600,000 |
| 5 | Unfinished | 2.132 | ~800,000 |

İkinci derinlikte (arkadaşların arkadaşları), ilişkisel veri tabanı ile graf veri tabanının performansları birbirine çok yakındır. Ancak üçüncü derinlikten (arkadaşlarının arkadaşlarının arkadaşları) itibaren, ilişkisel veri tabanının yaklaşık 30 saniyede tamamladığı görevi, Neo4j'nin graf veri tabanı 1,6 saniyeyle, neredeyse bir önceki sorgu süresine eşit olacak şekilde tamamlıyor. Sonraki derinliklere doğru ilerlendikçe ilişkisel veri tabanının tepki süresi kabul edilemez iken, graf veri tabanının tepki süresinin (1,3 ve 2,1 saniyeyle) kararlılığını koruduğu görülüyor.

## 2.7. Doğal Graf İşleme

İndekssiz komşuluk (index-free adjacency) özelliğine sahip graf veri tabanları doğal graf işleme yeteneğine sahiptir. İndekssiz komşuluk özelliğine sahip bir veri tabanı motorunda, her düğümün komşu düğümlere doğrudan referansları vardır. Bu yüzden her düğüm daha maliyetli olan global indeks (global index) kullanmak yerine, yakınındaki diğer düğümlerin bir mikro dizini gibi davranır. Bunun sonucu olarak sorgulama sürelerinin, grafın toplam boyutundan bağımsız olduğu, sadece aranan alt grafın boyutuyla orantılı olduğu anlamına gelir.

Doğal olmayan (nonnative) bir graf veri tabanı motoru ise düğümleri birbirine bağlamak için global indeksler kullanır. Bu indeksler her geçişe bir dolaylı katman ekler ve böylece daha yüksek hesaplama maliyetine neden olur. Doğal graf işlemeyi savunanlar hızlı ve verimli graf geçişleri için indekssiz komşuluğun çok önemli olduğunu savunurlar.



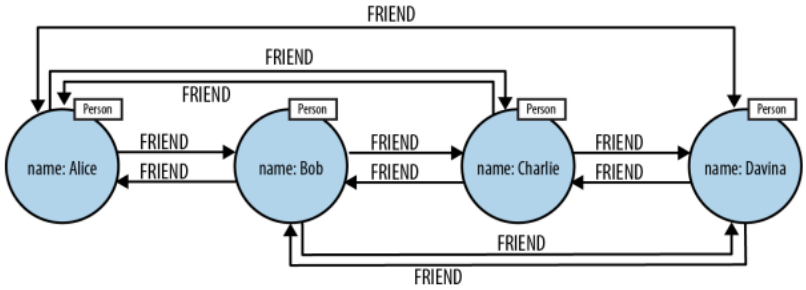

Şekil 2.6. İndeksleme kullanan doğal olmayan graf işleme,
(Robinson ve diğerleri, 2015: 150)

Doğal graf işlemenin, yoğun indekslemeye dayalı graflardan neden çok daha verimli olduğunu anlamak için onların çalışma zamanları incelenebilir. Veri tabanına dayalı verimli sayılabilecek indeks aramaları, (n veri tabanında yer alan kayıt sayısı olmak üzere) O(log n) algoritmik zamanlı veya en verimli aramayı ifade eden O(1) sabit zamanlı olabilir. m adımlık bir ağda gezinmek için indeksli yaklaşımın en kötü çalışma zamanı O(m log n) iken, indekssiz komşuluk yaklaşımınınki ise O(m) maliyetine sahiptir. Örneğin Şekil 2.6'da, indeksli yaklaşıma göre Alice'in arkadaşlarını bulmanın maliyeti O(log n) iken, kimlerin Alice ile arkadaş olduğunu bulmanın maliyeti, Alice ile potansiyel olarak arkadaş olan her bir düğüm için ayrı ayrı indeks araması yapılması gerektiğinden, O(m log n) olur (Robinson ve diğerleri, 2015: 149-151).

Yukarıdaki örnekte de işaret edildiği gibi sabit çalışma zamanının maliyeti (O(1)), logaritmik çalışma zamanının maliyetine (O(log n)) kıyasla çok daha az olduğundan; indekssiz komşuluk özelliğine sahip veri tabanı motorları, global indeksleme yapan graf veri tabanı motorlarına göre çok daha verimli çalışırlar.

Diğer taraftan ilişkileri simüle etmek için kullanılan (global) indeksler, indekslerin oluşturulduğu yönün tersine doğru hareket edilmesi söz konusu olduğunda, geriye doğru ilave indeksleme veya O(n) çalışma zamanına sahip (yukarıdan aşağıya tek tek tablonun tüm satırlarının kontrol edilmesine işaret eden) genel (brute-force) arama yapılmasını gerektirir.

İndeksli sorgular, küçük ağlar için çalışmakla birlikte büyük ölçekli graf veri tabanları üzerindeki sorgular için çok maliyetlidir. Doğal graf işleme yeteneklerine sahip graf veri tabanları, sorgu zamanında ilişkilerin rolünü ortaya çıkarmak için indeksleme aramalarını kullanmak yerine, yüksek performanslı geçişler sağlamak için indekssiz komşuluk kuralını kullanır (Robinson ve diğerleri, 2015: 151).




Şekil 2.7. İndekssiz komşuluk kuralını kullanan doğal graf işleme,
(Robinson ve diğerleri, 2015: 152)

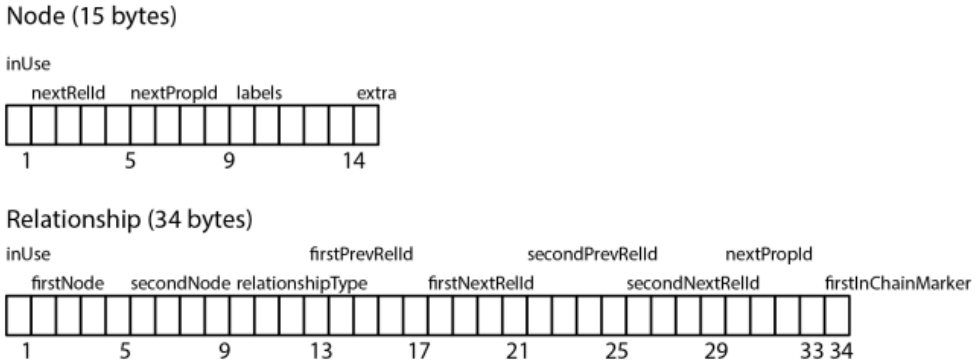
Şekil 2.7, kenarların indeksleme ihtiyacını nasıl ortadan kaldırdığını göstermektedir. Çift yönlü kenarlara (ilişkilere) sahip temsili grafta, Alice'in arkadaşlarını bulmak için her biri O(1) çalışma zamanına sahip ondan giden “arkadaş” kenarlarını (oklarını) takip etmek yeterlidir. Bu ilişkinin tersini ifade eden kimlerin Alice ile arkadaş olduğunu bulmak için (yine) her biri O(1) çalışma zamanına sahip ona gelen “arkadaş” kenarları (okları) takip edilir. Bu maliyetler göz önüne alındığında, büyük ölçekli veri tabanlarında graf geçişlerinin çok verimli olduğu söylenebilir.

## 2.8. Doğal Graf Depolama

Yüksek performanslı gezinmelere, sorgulara ve yazmalara olanak tanıyan indekssiz komşuluk kuralına (index-free adjacency) ilave olarak, graf veri tabanı tasarımının diğer önemli bileşenlerinden biri de graf depolama (graph storage) teknolojisidir. Graf algoritmaları için son derece hızlı gezinmeleri destekleyen etkin bir doğal graf depolama formatı, graf veri tabanlarını kullanmak için önemli bir sebep sayılabilir (Robinson ve

diğerleri, 2015: 153). Doğal graf depolama teknolojisini detaylı olarak incelemek amacıyla, Neo4j'ün mimarisi örnek olarak alınmıştır.

Neo4j mimarisinde, düğümler, kenarlar (ilişkiler), etiketler ve özellikler gibi grafın farklı bölümlerinin verileri farklı depolama dosyalarında saklanır.




Şekil 2.8. Düğüm ve kenar kayıtlarının depolandığı dosya yapısı, (Robinson ve diğerleri, 2015: 153)

Şekil 2.8'de gösterildiği gibi düğüm depolama dosyasında, sadece düğüm kayıtları depolanır. Grafın her düğümü, (söz gelimi) dokuz baytlık sabit boyutlu kayıtlar halinde diskte / depolama biriminde saklanır. Her birinin tekil kimliği olacak şekilde 100 düğümden oluşan bir graf düşünüldüğünde, kimliği 50 olan düğümün, düğüm depolama dosyasında (451 ve 459 dahil olmak üzere) 451 ile 459 baytları arasında saklandığına işaret eder. Benzer şekilde kimliği 10 olan düğümün, (her düğümün dokuz baytlık sabit boyutlu olduğu göz önüne alınarak) düğüm depolama dosyasının 91 ile 99 baytları arasında saklandığı anlaşılır. Böylece sabit boyutlu kayıtlar, maliyeti O(log n) olan bir arama yapmak yerine O(1) maliyetiyle bir kaydın konumuna doğrudan erişmeye olanak tanıdığından, depolama dosyasındaki düğümlere hızlı erişimi garanti eder.

Dokuz baytlık bir düğüm kaydının ilk baytı, düğümün kullanımda olup olmadığına işaret eden bayrak (flag) işaretçisine ayrılır. Sonraki dört bayt, düğüme bağlı (ilk) ilişkinin kimliğini; sonraki dört bayt, düğümün (ilk) özelliğinin kimliğini temsil eder. Düğümün etiketleri için beş bayt ayrılır. Son bayt ise yoğun şekilde ilişkili düğümleri tanımlamak amacıyla ilave bayraklar için ayrılmıştır (Bkz. Şekil 2.8).

Benzer şekilde ilişki depolama dosyasında, sadece kenar kayıtları depolanır. Her ilişki kaydı düğüm kaydında olduğu gibi sabit boyutlu kayıtlardan oluşur. İlişkinin; (1) türüne / yönüne,

(2) başındaki ve sonundaki her iki düğümün kimliklerine, (3) her başlangıç ve bitiş düğümünün önceki ve sonraki ilişki kayıtlarına ve (4) sonraki düğümün özelliklerine yönelik işaretçileri içerir. Bayrak ise genellikle ilişki zinciri (relationship chain) olarak adlandırılan düğümün ilk ilişki kaydının olup olmadığı bilgisini saklar (Bkz. Şekil 2.8). İlişki kayıtlarının, düğüm kayıtlarında olduğu gibi sabit boyut avantajına sahip olmasının yanı sıra çift yönlü olarak depolanması, veri tabanında tersine doğru hızlı bir şekilde gezinmeyi kolaylaştırır.

O halde doğal graf depolama teknolojisiyle belirli bir düğümden diğerine geçmek için düğüm kaydının kimliğiyle boyutunu çarpmak yeterli olacaktır. Benzer şekilde belirli bir ilişkiden sonraki düğüme geçmek için ilişki kaydı içinde yer alan sonraki (ikinci) düğüm kimliğiyle düğüm boyutunu çarpmak, doğrudan sonraki düğüme geçişi garanti edecektir. Benzer yöntemler kullanılarak geçişleri belirli ilişki türleri ile sınırlandırmak için ilişki türü alanına (relationship type), belirli etiketlerle sınırlandırmak için etiket alanına başvurulur.

Özellik grafına sahip veri tabanlarında, graf yapısının düğüm ve ilişki depolarına ek olarak, kullanıcının verilerini anahtar / değer çiftleri şeklinde saklayan özellik deposu dosyaları vardır. Özelliklerin (anahtar-değer çiftlerinin) işaretçi (bayrak) kayıtları düğüm ve ilişki kayıtlarına da eklendiğinden, özellik kayıtlarına düğüm ve ilişki kayıtlarından da erişilebilir.

Düğüm ve ilişki depolarında olduğu gibi özellik kayıtları da sabit bir boyuttadır. Her özellik kaydı dört özellik bloğundan ve özellik zincirindeki (yalnızca) bir sonraki özelliğin kimliğinden oluşur ve bir ile dört özellik bloğunda tutulur. Bu nedenle bir özellik kaydı en fazla dört özellik içerebilir. Özellik kaydı, özellik türüne ve özellik adının depolandığı özellik dizin dosyasına yönelik bir işaretçiyi içerir.

Her özellik değerine ilişkin kaydın, dinamik veya hizalanmış (inlined) değer işaretçisi vardır. Dinamik kayıtlar, büyük özellik değerlerinin depolanmasını sağlar. Dinamik dize (string) ve dinamik dizi (array) kaydı olmak üzere iki dinamik kayıt türü vardır. Dinamik kayıtlar, sabit boyutlu kayıtlarla ilgili listeleri içerir. Bu nedenle, çok büyük bir dize veya dizi birden fazla dinamik kayıtta tutulabilir (Robinson ve diğerleri, 2015: 156-157).

## 2.9. Fonksiyonel Olmayan Karakteristikler

Bir veri depolama teknolojisinin güvenilir olarak değerlendirilmesi, depolanan verilerin dayanıklılığı ve erişilebilirliği konusunda sağladığı garantiye dayanır. İlişkisel veri tabanlarının geleneksel olarak değerlendirildiği yaygın bir ölçü, işlemlerin (transactions) ACID (Atomicity, Consistency, Isolation, Durability) özelliklerinin sağlanması koşuluyla, işleyebilecekleri saniye başına işlem sayısıdır. İşlemlerin ACID özellikleri aşağıdaki gibi özetlenebilir.

*Bütünlük (Atomicity):* İşlemlerin veri tabanında bir bütün olarak değerlendirilmesidir. İşlem sırasında birden fazla veri tabanının güncellenmesi söz konusu olduğunda, tümünün birden başarılı olmasını veya başarısız olmasını ifade eder. Örneğin veri tabanlarının herhangi birinde hata oluşması durumunda, başarılı kayıtların gerçekleştirildiği veri tabanlarındaki işlemler de iptal edilir ve işlem tümüyle geçersiz olarak değerlendirilir.

*Tutarlılık (Consistency):* Veri tabanında kayıtlı verinin geçerliliğinin, işlem sonucunda, bir sonraki geçerli duruma geçmesidir. İşlem tam anlamı ile gerçekleşinceye kadar işlemden etkilenen verilerin bir önceki geçerli değeri saklanmaya devam edilir.

*Bağımsızlık (Isolation):* Eş zamanlı gerçekleşen işlemlerin birbirlerinden etkilenmemesi amacıyla işlemlerin birbirinden yalıtılmış, seri olarak yapılmasını ifade eder. Diğer bir ifadeyle işlem, başarılı ve başarısız olarak gerçekleştirilinceye kadar işlemden etkilenecek veri setlerinin birbirinden yalıtılmasını temsil eder.

*Dayanıklılık (Durability):* İşlem sırasında fiziksel veya işlemsel bir hata olması durumunda sistemin kendisini bir önceki geçerli veri durumuna döndürebilmesini ifade eder.

Büyük ölçekli verilerin kesintisiz işlenmesi ve yönetilmesi için ilişkisel bir veri tabanının ölçeklenmesi beklenir. Ölçeklemeyle birlikte sorguları ve güncellemeleri işlemek için birçok örnek kullanılabilir hale gelir. Böylece herhangi bir örneğin kaybedilmesi kümenin bir bütün olarak çalışmasını etkilemez.

Tüm graf veri tabanları tamamen ACID olmamakla birlikte, tutarlılığı garanti etmeleri, çökmelerden kurtulmaları ve veri bozulmasını önlemeleri gerekir. Ayrıca yüksek kullanılabilirlik ve performans sağlamak için ölçeklendirilebilir olmaları beklenir.

### 2.9.1. İşlemler

Tüm NoSQL veri tabanları işlemsel (transactional) olmasa da birçok graf veri tabanı (özellikle Neo4j özelinde), ilişkisel veri tabanı işlemleriyle anlamsal olarak aynıdır. Yazma işlemleri, tutarlılığın sağlanması amacıyla işlemle ilgili tüm düğümlerde ve ilişkilerde kilitlenerek gerçekleştirilir. İşlemin başarıyla tamamlanması durumunda dayanıklılık için değişiklikler diske aktarılır ve yazma kilitleri serbest bırakılır. Bu operasyonlar işlemin atomize (bütünlük) özelliğini garanti eder. İşlem herhangi bir nedenle başarısız olursa, (diske aktarılmadan önce) yazmalar iptal edilir ve yazma kilitleri serbest bırakılarak graf önceki tutarlı durumuna döndürülür. İki veya daha fazla işlem aynı graf elemanlarını eş zamanlı olarak değiştirmeye çalışırsa, olası bir kilitlenme durumu algılanarak işlemleri seri (sıralı) hale getirir. Böylece bir işlemin yazma operasyonları, diğer işlemler tarafından görülemeyeceğinden yalıtım (isolation) sağlanmış olacaktır (Robinson ve diğerleri, 2015: 162-163).

### 2.9.2. Kurtarılabilirlik

Kurtarılabilirlik (recoverability), veri tabanının, bir arıza ortaya çıktıktan sonra işleri ayarlama yeteneği ile ilgilidir. Herhangi bir sebeple (hatalı kapatma, donanımsal arızalar vb.) hatayla karşılaşan graf veri tabanı, hatalı kapanmadan kurtulurken, en son aktif işlem günlüğünü kontrol eder ve kayıtlarda bulduğu tüm işlemleri yeniden yürütür. Bunlardan bazılarının kaydı zaten başarılı bir şekilde uygulanmış olabilir. Ancak aralarında başarılı olmayan kayıtlar da olabilir. Bu yüzden son aktif işlem günlüğündeki kayıtlara göre tüm işlemler yeniden çalıştırılarak tutarlılık sağlanmış olur.

Bununla birlikte istemci uygulamalarına yüksek kullanılabilirlik sağlamak için kümeler halinde çalıştırılan veri tabanları, örneklerin kurtarılmasına ek avantajlar sağlar. Böylece yukarıdaki doğal kurtarma tamamlandıktan sonra, bir kopya, kümenin diğer üyelerinden (genellikle ana üyeden) yeni işlemler isteyebilir. Daha sonra bu yeni işlemleri, işlem tekrarı yoluyla kendi veri kümesine uygulayabilir (Robinson ve diğerleri, 2015: 163-164).

### 2.9.3. Kullanılabilirlik

Kullanılabilirlik (availability) veri tabanının çökme sonrasında bir bulut sunucusunu tanıma ve gerekirse onarma yeteneği, verilerin insan müdahalesi olmadan hızlı bir şekilde tekrar kullanılabilir hale gelmesi anlamına gelir.

Yüksek kullanılabilirlik için veri tabanı örnekleri kümelenir. Grafın tam bir kopyasının her makinede saklanmasını sağlamak için bir ana küme düzenlemesi kullanır. Yazmalar, ana bilgisayardan ana kümelere sık aralıklarla kopyalanır. Bu süreç bir veri tabanının dayanıklılığını arttırır. Ayrıca yazma işlemlerinde asimptotik ölçeklenebilirliğe, okuma işlemlerinde ise neredeyse doğrusal ölçeklenebilirliğe izin verir.

Kullanılabilirliğin başka bir yönü, kaynaklara erişim için çekişmedir. Grafın belirli bir bölümüne özel erişim (örneğin, yazmalar için) için başvuran bir işlem, yüksek gecikme süresine sahip olabilir. Graflarda erişim kalıpları, özellikle deyimsel doğal graf sorgularının yürütüldüğü yerlerde, daha eşit bir şekilde yayılma eğilimindedir. Bunun sonucu olarak genel sorgu yükünün düşük çekişmeli olarak dağıtılması (daha hızlı sorgu yapabilme) olanağına sahip olunur (Robinson ve diğerleri, 2015: 164-166).

### 2.9.4. Ölçeklenebilirlik

Ölçeklenebilirlik (scalibility) tüm büyük ölçekli verilerde önemli bir sorundur. Mapreduce tabanlı dağıtık / yatay ölçekleme (scaling out), dikey ölçekleme (scaling up) yöntemine göre her bir makinede daha zayıf bir performans sunar. Dikey ölçekleme yöntemi, önbellek performansının optimize edilmesini, çoklu çekirdeklerin kaldıraç etkisinden yaralanmayı ve paylaşılan büyük bellek düğümlerini kullanmayı işaret eder (Xia, Tanase, Nai, Tan, Liu, Crawford ve Lin, 2014).

Günümüzde veri hacimleri büyüdükçe ölçek konusu daha önemli hale gelmektedir. İlişkisel veri tabanlarıyla baş edilmesi zor olan büyük ölçekli veri sorunları, yüksek ölçeklenebilirliğe sahip graf veri tabanlarıyla çözülebilir. Bu sebeple graf veri tabanları modern uygulamaların yüksek ölçekli iş yükü taleplerini karşılamak için iyi bir alternatif olarak öne çıkar.

Ölçekleme, temel olarak üç parametreyle ölçülen toplam bir değerdir. Bunlar grafın boyutunu ifade eden kapasite, tepki süresini temsil eden gecikme ve zamana göre işlenen

veri hacmini yansıtan okuma ve yazma verimliliği kriterleri olarak ifade edilebilir (Robinson ve diğerleri, 2015: 166).

Kapasite

Bazı graf veri tabanı sağlayıcıları, performans ve depolama maliyeti karşılığında graf boyutunda herhangi bir üst sınırdan kaçınmayı tercih ederken, bazıları graf boyutlarını optimize ederek daha hızlı performans ve daha düşük depolama maliyetini tercih etmektedir. Çünkü grafın boyutu ile performans ve depolama maliyetleri arasında bir değiş-tokuş (tradeoff) vardır. Bu değiş tokuşun nedeni, "Doğal Graf Depolama" başlığı altında ele alınan depolama birimleri içinde yoğun bir şekilde kullanılan sabit kayıt boyutlarından ve işaretçilerden kaynaklanmaktadır.

Graf veri tabanının sunduğu avantajlardan yararlanmak için belirli bir kapasiteye (capacity) ulaşmış olması gerekir. Bu sınır graf sağlayıcılarına göre değişmekle birlikte, ikinci ve üçüncü derece sorgular için birkaç bin düğüme sahip veri kümeleri başlangıç olarak kabul edilebilir. Diğer taraftan veri hacminden bağımsız olarak sorgunun derecesi arttıkça da graf veri tabanlarının performans avantajları daha açık görülür. Graf veri tabanlarının anlamlı uygulanabilirlik aralığı olarak on binlerce düğüm ve yüz binlerce ilişkiden, milyarlarca düğüm ve ilişkiye kadar değişebilen geniş bir aralığa işaret edilir (Robinson ve diğerleri, 2015: 167).

Gecikme

Graf veri tabanları, veri hacminin artmasına paralel "join" işlemlerinin uzun sürdüğü ilişkisel veri tabanlarının paylaştığı gecikme (latency) sorunlarına sahip değildir.

Bir graf veri tabanında birçok sorgu, bir başlangıç düğümünü (veya düğümlerini) bulmak için bir indeksin kullanıldığı bir modeli takip eder. Sorguyu gerçekleştirmek için geçişlerin geri kalanında ("Doğal Graf Depolama" başlığı altında detaylı olarak açıklanan) işaretçi izleme ve model eşleştirme kombinasyonu kullanılır. Böylece graf veri tabanlarında sorgulama performansı, ilişkisel veri tabanlarından farklı olarak veri setinin boyutundan bağımsız olur. Diğer bir ifadeyle graf veri tabanlarında sorgulama, veri kümesinin boyutu

büyüse bile neredeyse sabit olan performans sürelerine yol açar (Robinson ve diğerleri, 2015: 167-168).

Verimlilik

Graf veri tabanları diğer veri tabanlarından farklı bir şekilde ölçeklenir. I / O yoğun uygulama davranışları incelendiğinde, karmaşık bir operasyon tipik olarak bir dizi ilgili veriyi okuyup yazar. Diğer bir ifadeyle uygulama, genel veri kümesi içinde mantıksal bir alt graf üzerinde okuma ve yazma şeklinde çoklu görevleri yerine getirir. Daha büyük ve birbirine daha bağımlı bu tür çoklu görevler graf veri tabanlarıyla kolaylıkla gerçekleştirilebilir. Ayrıca her bir işlemi eş anlı olarak doğal bir graf deposuyla yürütmek, aynı işlemi ilişkisel işlemle yürütmekten daha az hesaplama maliyetine sahiptir. Graflar, aynı görevi daha az hesaplama gerektirmesi için ölçeklenir.

Daha karmaşık görevler tek bir makinenin sorgu çalıştırma kapasitesini ve daha spesifik olarak I / O verimini aşacaktır. Bu durumda yüksek kullanılabilirlik ve okuma verimi için yatay olarak ölçeklemeye başvurulabilir. Okumaların yazmalardan çok daha fazla olduğu tipik graf iş yükleri için yatay ölçekleme (dağıtık yapı) mimarisi ideal bir çözüm olabilir.

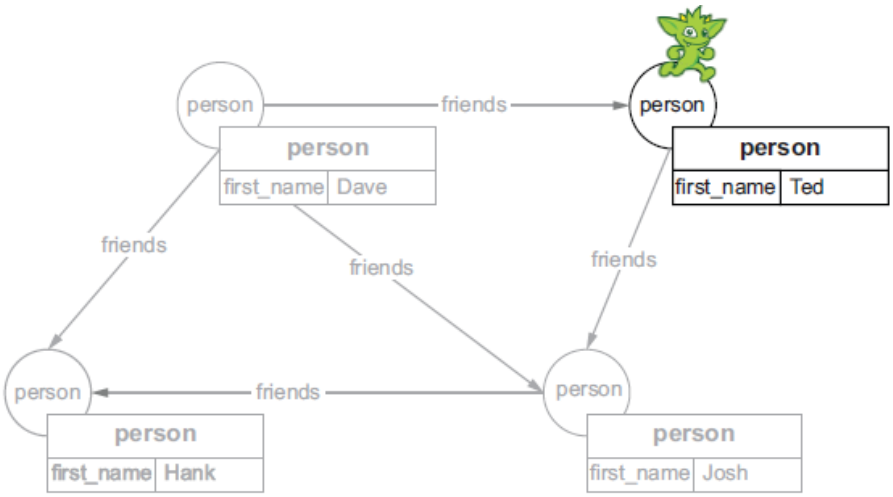
Bir kümenin kapasitesinin aşılması durumunda, uygulamada parçalama (sharding) mantığı oluşturularak bir graf farklı graflara bölünebilir. Parçalama, uygulama düzeyinde veri tabanlarının bölünmüş kayıtlarını birleştirmek için yapay bir tanımlayıcı kullanılmasını gerektirir. Bu yöntemin performansı büyük ölçüde grafın şekline bağlıdır. Bazı graflar için bu yöntem başarılı sonuçlar verebilmektedir. Örneğin Mozilla, yeni nesil bulut tarayıcısı Pancake'in bir parçası olarak Neo4j graf veri tabanını kullanmaktadır. Böylece tek bir büyük grafa sahip olmak yerine, her biri son kullanıcıya bağlı çok sayıda küçük bağımsız grafı depolamak, ölçeklendirmeyi çok kolaylaştırır (Robinson ve diğerleri, 2015: 168).

# 3. TEMEL GRAF İŞLEME TEKNİKLERİ

Gelişmiş graf işleme yöntemlerine esas teşkil eden temel graf işleme teknikleri, grafların veri işleme ve çalışma ilkelerini daha yakından görmek amacıyla, kavramsal ve uygulama düzeyinde incelenmiştir. Bu çerçevede uygulama dili olarak Apache Tinkerpop projesi kapsamında geliştirilen Gremlin graf sorgulama dili kullanılmıştır.

## 3.1. Graflarda Gezinme

Bir grafın düğümlerden kenarlara ya da kenarlardan düğümlere doğru adım adım taranması, gezinme (traversing) olarak adlandırılır.




Şekil 3.1. Graflarda gezinme, (Bechberger ve Perryman, 2020: 59)

Şekil 3.1'de gösterildiği gibi "Ted'in arkadaş olduğu kişiler kimler?" gibi bir sorunun yanıtı için öncelikle, başlangıç noktasını temsil eden Ted'in düğümü bulunur. Ted'in düğümünden Josh'ın düğümüne doğru olan "arkadaşlar" kenarı (edge) üzerinde hareket edilerek Josh'ın düğümüne ulaşılır. Her ne kadar Ted'in düğümüyle bağlantılı olsa da Dave'in düğümünden Ted'in düğümüne doğru olan kenarda ise hareket edilmez. Çünkü örneğe göre "Ted ile arkadaş olan kişiler kimler?" sorusunun değil, "Ted'in arkadaş olduğu kişiler kimler?" sorusunun yanıtı aranmaktadır. Bu örnekte sadece bir kenar üzerinde gezinme söz konusudur. Eğer gezinmeye Dave'in düğümünden başlansaydı, gezinme süreci çoklu paralel süreçlerden oluşurdu. Çünkü "Dave'in arkadaş olduğu kişiler kimler?" sorusunun yanıtı için Dave'in düğümünden Ted'in, Josh'un ve Hank'ın düğümüne doğru üç kenarda birden gezinme gerçekleşecektir.

Yukarıdaki örneklerden bir grafta gezinme sürecinin dört kritik karakteristik özelliği çıkartılabilir.

- Gezinme süreci, graf üzerinde hareket etmeyi ifade eden bir dizi adımdan oluşur. Bu adımlar, kenarlar üzerinde gezinme kadar veri filtreleme gibi graf verisini işleyen çeşitli operasyonlar da içerebilir. Her adım bir başlangıç konumundan başlar ve bitiş ya da hedef olarak adlandırılan farklı bir konumda sonlanır.
- Gezinme süreci, graf üzerindeki konumun bilinmesini gerektirir. Graf üzerinde gezinirken etkili bir yön tayini yapabilmek amacıyla, grafın yapısı içinde nerede olunduğunun takip edilmesi gerekir.
- Kenarların (edge) yönü önemlidir. İlişkisel veri tabanlarının aksine, graf veri tabanlarında tüm ilişkilerin yönü karşılıklı (iki yönlü) olmayabilir.
- Gezinmelerin hafızası yoktur. Diğer bir ifadeyle gezinme sırasında mevcut konum bilinmesine rağmen, önceki konum bilinmez.

### 3.2. Tekrarlı Gezinme

Tekrarlı gezinme (recursive / looping traversal), graf veri tabanlarının en güçlü özelliklerinden biridir. Gezinmenin ard arda birden fazla kez gerçekleştirilmesinin gerektiği problemler için tekrarlı gezinme kullanılır. Örneğin haritada iki konum verildiğinde, başlangıç konumundan bitiş konumuna gitmek için sokakların ve dönüşlerin sayısı önceden tahmin edilemez. Graf veri tabanları yüksek derecede birbiriyle bağlantılı verilerle baş etmek için optimize edildiğinden, onların sorgulama dilleri ve temel veri yapıları tekrarlı sorguları hızlı bir şekilde gerçekleştirebilir (Bechberger ve Perryman, 2020: 69).

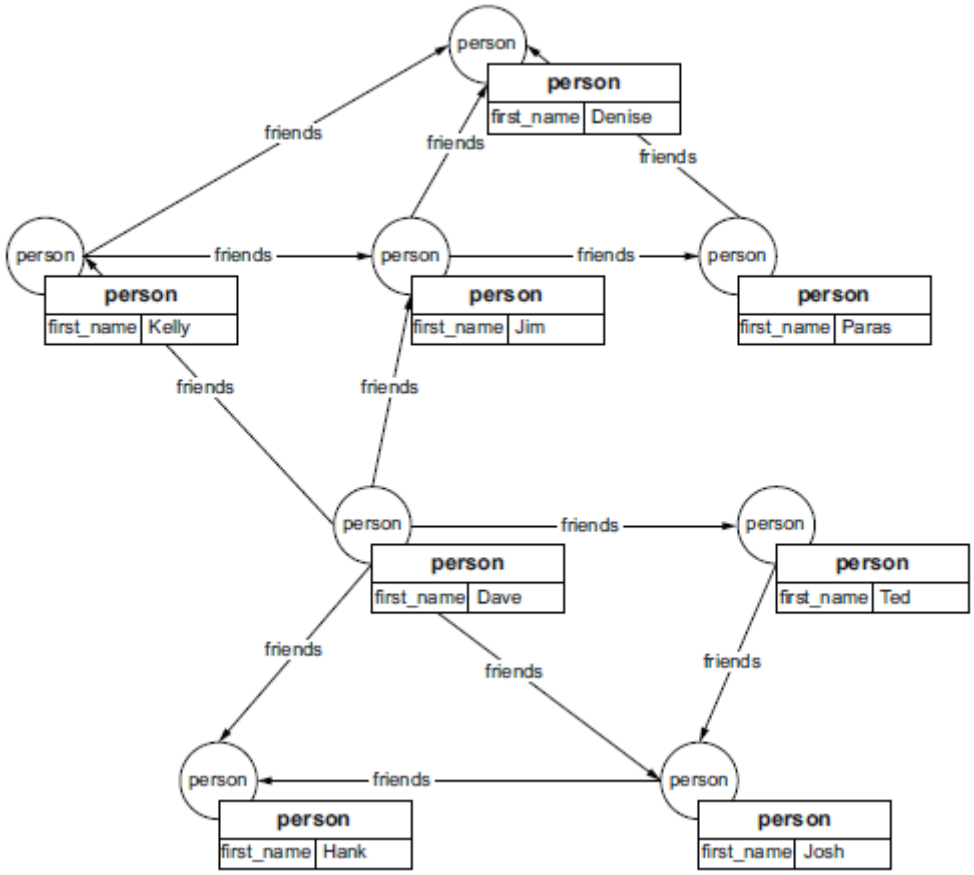
```
g.V().has('person', 'first_name', 'Ted').
repeat(out('friends')).times(2).
values('first_name')
==>Hank
```

Yukarıdaki Gremlin sorgusunun ilk satırı başlangıç konumu olarak Ted'in düğümünden başlanacağını ifade eder (Bkz. Şekil 3.1). İkinci satır iki kez gezinme yapılacağını gösterir. İlk adımda Ted'in düğümünden giden kenar / bağlantı üzerinde ilerlenerek Josh'un düğümüne, ikinci adımda ise ilk adımdaki sürecin Josh'un düğümünde tekrar edeceğini, dolayısıyla Josh'un düğümünden giden kenar / bağlantı üzerinde ilerlenerek Hank'in

düğümüne varılacağını ifade eder. Üçüncü satır ziyaret edilen son düğümünün değerinin döndürüleceğini temsil eder. Dördüncü satır ise sorgu sonucunda döndürülen değerdir.

### 3.3. Graflarda CRUD Operasyonları

Graflarda veri ekleme, okuma, güncelleme ve silme gibi CRUD operasyonları patika bulma (path finding) işlemleri üzerinden gerçekleştirilir. Patika bulma işlemleri ve CRUD operasyonlarını göstermek amacıyla aşağıdaki örnek graftan yararlanılacaktır.




Şekil 3.2. Patika bulma ve CRUD operasyonları, (Bechberger ve Perryman, 2020: 82)

Yeni bir veriyi grafa eklemeyi ifade eden graf veri tabanında bir nesnenin yaratılması, uygun ögeleri ve özellikleri içeren yeni bir düğümün oluşturulmasını gerektirir. Şekil 3.2'de belirtilen temsili grafa Gremlin ile bir düğüm eklenmesine ilişkin kodlar aşağıda gösterilmektedir.

```
g.addV('person').property('first_name', 'Dave')
==>v[13]
```

“g” graf üzerinde hareket edilecek başlangıç noktasını, “addV(‘person’)” yeni düğümün türünün / etiketinin “person” olacağını, “property(‘first_name’, ‘Dave’)”ise anahtarın (key) “first_name” ve değerinin (value) “Dave” olduğunu ifade eder. Gremlin tarafından döndürülen bir değer olan “v[13]” ifadesi ise eklenen düğümün kimlik no.sunu (id) temsil eder. İşlemin grafa doğru bir şekilde eklendiğinin kontrolü aşağıdaki kodlarla teyit edilebilir.

```
g.V().has('person', 'first_name', 'Dave')
==>v[0]
==>v[13]
```

Gremlin’in “v[0]” ve “v[13]” şeklinde iki farklı id döndürmesi, grafta “person” etiketli ve özellik (property) değeri “Dave” olan bir düğümün zaten olduğuna, var olan düğümün silinerek yeni bir düğümün oluşturulduğuna işaret eder (Bkz. Şekil 3.2).

Graf üzerinde yapılan temel değişikliklerden bir diğeri de kenar (edge) eklemedir. Aşağıda Ted ve Hank düğümleri arasındaki bağlantıyı ifade eden bir “friends” (arkadaşlar) kenarının eklenmesi gösterilmektedir.

```
g.addE('friends').
from(V().has('person', 'first_name', 'Ted')).
to(V().has('person', 'first_name', 'Hank'))
==>e[15][4-friends->6]
```

“g” graf üzerinde hareket edilecek başlangıç noktasını, “addE('friends')” yeni kenarın türünün / etiketinin “friends” olacağını, “from(V().has('person', 'first_name', 'Ted')).” ile “to(V().has('person', 'first_name', 'Hank'))” ifadeleri ise bağlantının Ted düğümünden Hank düğümüne doğru olduğunu ifade eder. Gremlin tarafından döndürülen bir değer olan “e[15][4-friends->6]” ifadesi ise 4 no.lu düğümden 6 no.lu düğüme doğru 15 no.lu bir kenarın eklendiğini temsil eder.

Bir düğümün graftan silinmesi ise şu şekilde gerçekleştirilir.

```
g.V(13).drop()
```

Daha önce de belirtildiği üzere “g” graf üzerinde hareket edilecek başlangıç noktasını, “V(13)” işlem yapılacak düğümün id no.sunu, “drop()” ise söz konusu düğümün silinmesini ifade eder. Yukarıdaki komut herhangi bir değer döndürmediği için işlemi kontrol etmek için aşağıdaki kodlar kullanılabilir.

```
g.V().has('person', 'first_name', 'Dave')
==>v[0]
```

Yukarıdaki komutun döndürdüğü “v[0]” değeri, “person” düğümlerinde “first_name” anahtarına sahip “Dave” değerinin graf veri tabanında olmadığını, yani bir önceki düğüm silme işleminin başarılı bir şekilde gerçekleştirildiğini ifade eder. Benzer şekilde kenar da silinebilir.

```
g.E(15L).drop()
```

Yukarıdaki komut id no.’su 15 olan kenarı (edge) graf veri tabanından silecektir. Graf veri tabanında güncelleme yapmak için aşağıdaki komutlar kullanılabilir.

```
g.V().has('person', 'first_name', 'Dav').
property('first_name', 'Dave')
==>v[18]
```

Yukarıdaki komut dizisi “person” düğümlerinde “first_name” anahtarına sahip “Dav” değerlerinin, “Dave” olarak değiştirileceğini ifade eder. Gremlin’in döndürdüğü “v[18]” değeri, söz konusu işlemin id no.’su 18 olan düğümde başarılı bir şekilde gerçekleştirildiğini gösterir. Aşağıdaki iteratif komut ise graf veri tabanında yer alan tüm düğümleri herhangi bir değer döndürmeden siler.

```
g.V().drop().iterate()
```

Bir düğüm yeniden kullanılabilir şekilde oluşturulmak istenirse, aşağıda gösterildiği gibi bir değişken ile temsil edilebilir.

```
dave = g.addV(‘person’).property(‘first_name’, ‘Dave’).next()
ted = g.addV(‘person’).property(‘first_name’, ‘Ted’).next()
hank = g.addV('person').property('first_name', 'Hank').next()
```

```
josh = g.addV('person').property('first_name', 'Josh').next()
```

“dave” ve “ted” ifadeleri Dave’e ve Ted’a ait düğümleri temsil etmek üzere, Dave’in düğümünden Ted’in düğümüne doğru, etiketi “firends” olan bir kenar (edge) aşağıda gösterildiği gibi eklenebilir.

```
g.addE('friends').from(dave).to(ted).next()
g.addE('friends').from(dave).to(hank).next()
```

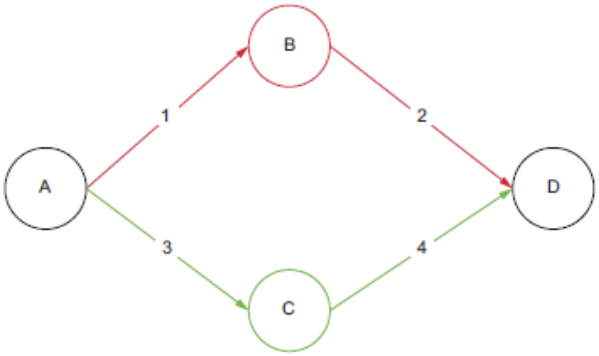
Yukarıdaki komutlarda düğümlerin (vertex) ve kenarların (edge) aynı anda oluşturulduğu, diğer bir ifadeyle düğüm komutlarının değişkenlere atanma sırasında grafta herhangi bir düğüm oluşturulmadığı belirtilmelidir. Benzer açıklamalar Hank’a ait düğüm ve kenar için de geçerlidir. “iterate()” ve “next()” ifadeleri aynı işlevlere sahip olmasına rağmen, “iterate()” ifadesi gezinmenin sonucuna ilişkin herhangi bir değer döndürmezken, “next()” ifadesi döndürür.

Graf veri tabanlarında bir değişiklik birden fazla öge ile eş anlı bir şekilde yapılabilir.

```
g.addE('friends').from(dave).to(josh).
addE('friends').from(dave).to(hank).iterate()
```

### 3.4. Graflarda Patikalar

Patikalar (paths), graf üzerinde başlangıç düğümünden bitiş düğümüne ulaşmak için yapılan gezintinin adımlar dizisini ifade eder. Patikalar, sadece bağlantılı iki düğümün tespit edilmesini değil, aynı zamanda başlangıç düğümünden bitiş düğümüne nasıl varıldığının da kesin olarak belirlenmesine olanak tanır. Çünkü başlangıç ve bitiş düğümleri arasında birden fazla farklı yol olabilir.




Şekil 3.3. A'dan D'ye giden iki farklı patika örneği,
(Bechberger ve Perryman, 2020: 98)

A– 1 → B – 2 → D

A– 3 → C – 4 → D

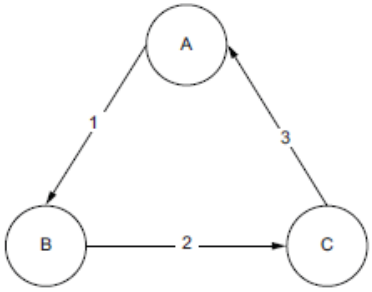
Şekil 3.3'te A düğümünden D düğümüne iki farklı patika vardır. Birinci patika A düğümünün 1 no.lu kenarından B düğümüne, oradan da B düğümünün 2 no.lu kenarından D düğümüne götüren adımlar dizisidir. İkinci patika ise A düğümünün 3 no.lu kenarından C düğümüne, oradan da C düğümünün 4 no.lu kenarından D düğümüne götüren adımlar dizisidir. Yukarıdaki patikanın sorgusu iteratif yolla aşağıda gösterildiği gibi oluşturulabilir

```
g.V().has('person', 'first_name', 'Ted').
until(has('person', 'first_name', 'Denise')).
repeat(both('friends')).path()
```

"path()" ifadesi, gezinme devam ederken ziyaret edilen düğümlerin (ya da opsiyonel olarak kenarların) geçmişini döndürür. Yukarıdaki sorgunun çalıştırılması sonucunda sonsuz bir döngüye (cycle) girileceğinden Gremlin aşağıdakine benzer bir hata değeri döndürecektir.

```
Script evaluation exceeded the configured
'scriptEvaluationTimeout' threshold
of 30000 ms or evaluation was otherwise cancelled directly
for request
[g.V().has('person', 'first_name', 'Ted').
until(has('person', 'first_name', 'Denise')).
repeat(both('friends')).path()]
Type ':help' or ':h' for help.
```

Yukarıdaki hataya yol açan ve döngü (cycle) olarak bilinen kavram, aşağıdaki şekilde gösterildiği gibi her biri birer varış noktası olan bir veya daha fazla düğüme sahip bir graftaki düğüm ve kenarlardan oluşan bir patikadır.




Şekil 3.4. Başlangıç düğümü A'dan tekrar A'ya gelen bir döngü, (Bechberger ve Perryman, 2020: 100)

Şekil 3.4'te yer alan temsili grafta A, B ve C düğümlerinin tümü, aşağıdaki adımları izleyerek birbirine benzer gezinti paternlerini kullanır.

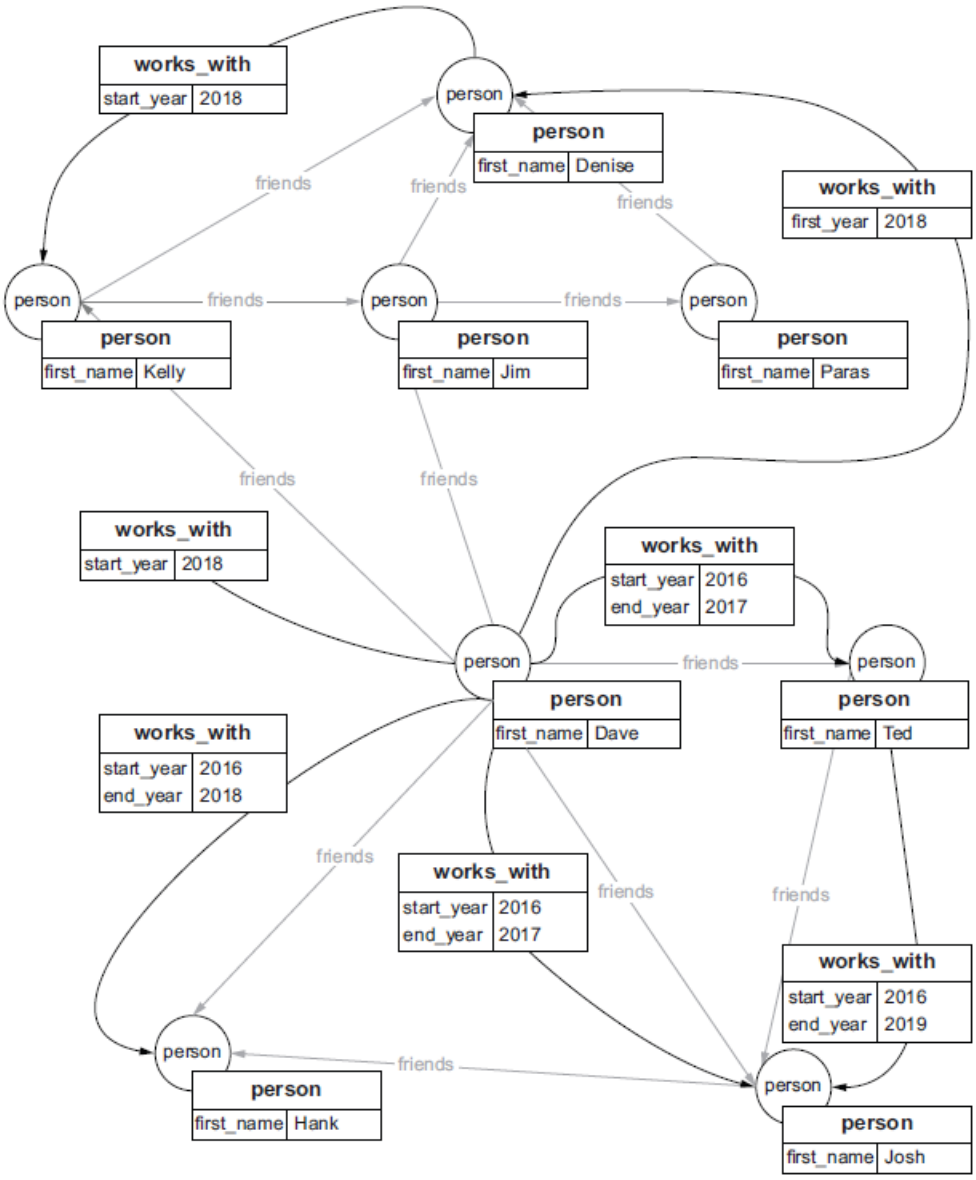
[A → 1 → B → 2 → C → 3 → A]

[A → 3 → C → 2 → B → 1 → A]

Söz konusu döngülerden (cycle) kurtulmak amacıyla, patikalar arasındaki en basit yolu ifade eden "basit patika" (simple path) kavramından yararlanılır. Bu çerçevede basit patika, herhangi bir düğümde tekrarlayan bir gezinti döngüsüne girmeden, tüm düğümlerin sonuçlarını veren yol olarak tanımlanabilir. Bir düğümde daha önce gezinildiği bilgisi yardımıyla, döngüye girilmeden düğümler arasında dolaşılabilir. Bir düğümde daha önce gezinildiği bilgisine sahip bir algoritma, mevcut adımın bir döngü olduğunu tespit ederek gezinmeyi sonlandırır. Bu amaçla Gremlin sorgulama dilinde "simplePath()" ifadesi kullanılır. "simplePath()", aynı düğümü birden fazla kez ziyaret eden adımları filtreleyerek gezinmeyi gerçekleştirir (Bechberger ve Perryman, 2020: 101-102). Yukarıda hata döndüren sorguya, aşağıda gösterildiği gibi "simplePath()" ifadesi ilave edilir.

```
g.V().has('person', 'first_name', 'Ted').
until(has('person', 'first_name', 'Denise')).
repeat(both('friends').simplePath()).path()
```

Yukarıdaki sorgu, herhangi bir hata döndürmeksizin Ted'in düğümünden Denise'in düğümünü giden alternatif düğümlerin listesini çıkaracaktır. Ancak patikalar, sadece düğümleri değil, aynı zamanda kenarları da içerir. Dolayısıyla patikanın diğer bir parçası olan kenarı elde etmek için düğüm-kenar-düğüm yolu izlenir. Örnek olarak kullanılan "friends" bağlantısının yer aldığı grafa, insanların birlikte çalıştığı başlangıç ve bitiş yılları arasındaki bağlantıyı sağlayan "works_with" isimli yeni bir ilişki eklenebilir (Bechberger ve Perryman, 2020: 103). "works_with" isimli yeni ilişkinin eklendiği graf Şekil 2.5'te gösterilmektedir.



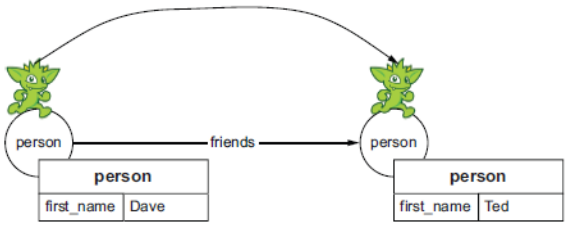

Şekil 3.5. Temsili grafın "works_with" isimli kenarlar ile genişletilmesi, (Bechberger ve Perryman, 2020: 104)

Amaç düğümlerde gezinip, (kenar) özelliğine (property) göre filtrelenen kenarlarda durmak ve sonra komşu düğüme geçmektir. Kenarlarda gezinmeyi ve filtrelemeyi temsil eden bu adımları gösteren Gremlin sorgusu aşağıda sunulmaktadır.

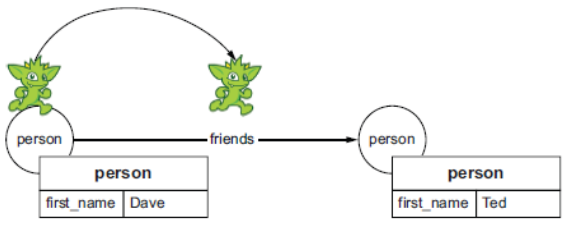
```
g.V().has('person' ,'first_name', 'Dave').
bothE('works_with').has('end_year',lte(2018)).
otherV().values('first_name')
```

Daha önce belirtildiği gibi "g" graf üzerinde hareket edilecek başlangıç noktasını, "V()" graftaki tüm düğümleri ki bu durumda, "V()has('person' ,'first_name', 'Dave')" ifadesi adı Dave olan graftaki tüm kişileri temsil eder. "bothE('works_with').has('end_year',lte(2018))" ifadesi yönü dikkate alınmaksızın "works_with" bağlantısına ait kenarların 2018 ve daha önceki yıllara göre filtrelenmesini işaret eder. "otherV()" ise kenardan ayrılarak diğer bir düğüme geçilmesini, "values('first_name')" ise sorgunun döndüreceği değeri ifade eder.

Yönü bir düğüme doğru olan (gelen) kenarlar "inE()", bir düğümden çıkan (giden) kenarlar gezinme "outE()" ve kenarın yönü dikkate alınmaksızın düğüme gelen ve düğümden giden kenarlar üzerinde çift yönlü gezinme ise "bothE()" sorgusuyla gerçekleştirilir. "in()", "out()" ve "both()" sorguları da benzer işlevi yerine getirir. Ancak gezinme, "inE()", "outE()" ve "bothE()" gibi önünde "E" takısı olan sorgularda kenarlarda, "in()", "out()" ve "both()" gibi önünde "E" takısı olmayan sorgularda ise düğümlerde sonlanır.




Şekil 3.6.a. Gezinmenin düğümde sonlanması, "out()", (Bechberger ve Perryman, 2020: 106)




Şekil 3.6.b. Gezinmenin kenarda sonlanması, "outE()", (Bechberger ve Perryman, 2020: 106)

Aşağıdaki sorgu, gezinmeyi düğümde sonlandırır (Şekil 3.6.a).

```
g.V().has('person', 'first_name', 'Dave').out()
```

Aşağıdaki sorgu ise gezinmeyi kenarda sonlandırır (Şekil 3.6.b).

```
g.V().has('person', 'first_name', 'Dave').outE()
```

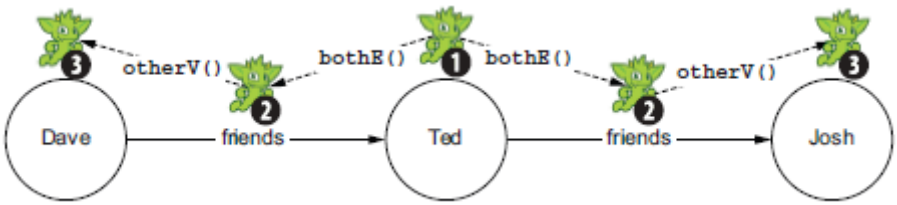
Adı Dave olan kişilerin birlikte çalıştığı kişilerin ismini getiren sorgu ise şu şekilde ifade edilebilir.

```
g.V().has('person', 'first_name','Dave').
both('works_with').values('first_name')
==>Ted
==>Josh
==>Hank
==>Kelly
==>Denise
```

Yukarıdaki sorguda yer alan "==>" ile başlayan satırlar sorgunun sonucunu ifade eder (Bkz. Şekil 3.5). Kenar özelliklerine (edge properties) göre filtreleme, zamana ve ağırlığa dayalı (time-based ve weight-based) olarak genellikle iki kategoride incelenir. Zamana dayalı kenar özeliğine göre filtrelemeye, Dave'in 2018 ve ondan önceki yıllarda birlikte çalıştığı kişilerin listelenmesi örnek olarak verilebilir eder (Bkz. Şekil 3.5).

```
g.V().has('person','first_name','Dave').
bothE('works_with').has('end_year',lte(2018)).
otherV().values('first_name')
==>Josh
==>Ted
==>Hank
```

Yukarıdaki sorguda yer alan "lte" ifadesi, "less than or equal to" ifadesinin kısaltması olup "küçük-eşit" anlamına gelir. "otherV()" ifadesi ise bir önceki adımda yer alan düğümden diğer düğümlere doğru bir gezinmeyi temsil eder.




Şekil 3.7. "bothE()" ve "otherE()" sorgularının işleyişi, (Bechberger ve Perryman, 2020: 107)

“bothE()” ve ”otherV()” kombininde, bir düğümden diğer düğüme geçilirken kenarlarda yer alan etiket ve / veya anahtar-değer ikilileri de göz önünde bulundurulur.

```
g.V().has('person', 'first_name', 'Ted').
bothE().otherV()
```

Ted’in düğümünden önce “bothE()” ifadesiyle ok yönünde ve ok yönünün tersinde kenarlara doğru hareket edilir.“otherV()” ifadesiyle de (2 no.lu noktaların işaret ettiği) kenarladaki duraklamadan sonra, (Ted’in düğümünü işaret eden) gelinen düğümün tersi yöndeki düğümlere, yani Dave’in ve Josh’un düğümlerine geçilir (Şekil 3.7).

Patika sonuçlarına kenarları da dahil etmek için “bothE().otherV()” kombininden yararlanılabilir. Böylece gezinmelerde duraklama noktası olarak sadece düğümler değil, kenarlar da yer alır.

```
g.V().has('person', 'first_name', 'Ted').
until(has('person', 'first_name', 'Denise')).
repeat(bothE('works_with').otherV().simplePath()).path()
```

Bu yüzden bu tür bir sorgunun sonuçlarında sadece gezinilen düğümler değil, aynı zamanda kenarlar da bulunur. Yukarıdaki sorgunun döndürdüğü değerler (sonuçlar), diğer bir ifadeyle alternatif yollar aşağıda gösterilmektedir.

```
==>path[v[4], e[29][0-works_with->4], v[0], e[33][0-works_with->19],
v[19]]
==>path[v[4], e[29][0-works_with->4], v[0], e[32][0-works_with->13],
v[13], e[34][19-works_with->13], v[19]]
==>path[v[4], e[30][2-works_with->4], v[2], e[28][0-works_with->2],
v[0],e[33][0-works_with->19], v[19]]
==>path[v[4], e[30][2-works_with->4], v[2], e[28][0-works_with->2],
v[0], e[32][0-works_with->13], v[13], e[34][19-works_with->13], v[19]]
```

Bu tür bir sorgu düğümlerde havalimanı, kenarlarda ise uçuşların temsil edildiği bir grafta, hava trafik rotaları için iyi bir örnek uygulama olabilir. Böylece varış zamanı, kalkış zamanı, uçuş numarası, hava yolu şirketinin adı gibi kritik uçuş bilgileri elde edilebilir (Bechberger ve Perryman, 2020: 109).

### 3.5. Performans Optimizasyonu

Grafların temel çalışma ilkelerini ve performans optimizasyonunu anlamak için koridorlarla birbirine bağlı bir dizi odanın bir graf veri tabanına benzetildiği, kapıları üzerinde çekmeceleri olan bir graf "kaçış odası" metaforu incelenebilir. Bu benzetmede kapısı kapalı her oda birer düğümü temsil eder. Odanın içinden doğal olarak sadece odanın içindeki nesneler görülebilir. Odanın içindeki nesneler ise şunlardır:

- Her çekmecenin etiketlendiği bir şifonyer. Her çekmece bir düğüm özelliğini ve her etiket ise özellik ismini ifade eder.
- "Giriş" ve "çıkış" yazılı işaretleri ve etiketleri bulunan bir dizi kapı. Her kapı birer kenarı, (odadan çıkışı ya da odaya girişi gösteren) levhalar ise kenarların yönünü temsil eder.
- Kapıların da çekmeceleri vardır. Bu çekmeceler kenar özelliklerini ifade eder.

Odanın içinde yer alan görünür her şeye anında erişilebilir; düğüm özelliği, kenarlar ve kenar özellikleri. Ancak çekmecelerin ve kapıların ötesindeki (başka bir odadaki) herhangi bir şeye erişmek ilave çaba gerektirir; ilave adımları gerektiren gezinme süreci. Graf düğümünün mental modeli basit bir şekilde bir "kaçış odası" (escape room) metaforudur.

Düğüm özelliği, kenarlar ve kenar özellikleri bir düğümde konumlandırıldığı için bunları kullanmanın ilave bir maliyeti yoktur. Odanın dışındaki herhangi bir şeye, diğer bir ifadeyle başka bir düğüme, ancak bir kenar üzerinde gezinilerek erişilebilir. Odanın / düğümün dışındaki herhangi bir şeye erişim, önbellek taraması (a cache hit), disk işlemi (disk operation) ya da muhtemel bir graf ağı araması (network call) gibi ilave maliyetlere sahiptir. Bu ilave maliyetlerden dolayı mevcut düğüm üzerinde kalmaya çalışmak performans optimizasyonuna işaret eder (Bechberger ve Perryman, 2020: 62, 110). Kenarlar üzerinde gezinmeyi ifade eden "E" adımları, odaların kapılarını açmadan odalara bakmayı mümkün kılar.

```
g.V().bothE().count()
```

Yukarıdaki sorgu, başlangıç düğümünden hareket edilerek sadece kenarlar üzerinde gezinmeyi, diğer bir ifadeyle mevcut düğüm odasından görülen kenar kapılarını saymaya benzediğinden ilave maliyetlere sahip değildir.

```
g.V().both().count()
```

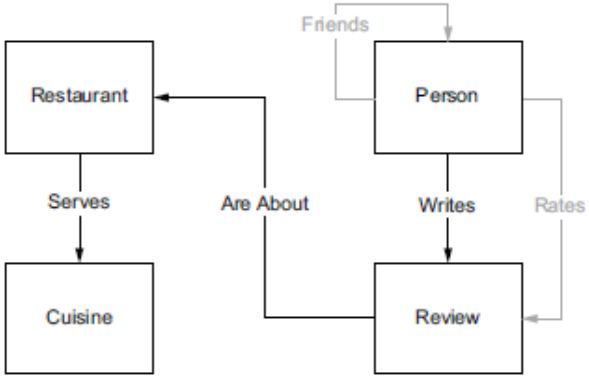
Yukarıdaki sorgu ise başlangıç düğümünden hareket edilerek düğümlerde gezinmeyi, diğer bir ifadeyle her kenar kapısından geçilerek diğer taraftaki düğüm odalarını saymaya benzediği için ilk yönteme göre daha maliyetlidir.

Odaları ya da kapıları saymanın sonucu aynı olduğundan, ikinci yönteme göre ilk yöntem çok daha performanslı bir gezinme olacaktır.

### 3.6. Bilinen Yürüyüş

Graf teorisinde yürüyüş (walk), bir dizi düğüm ve kenarı ifade eder. Patika ise farklı düğümlerde yapılan belirli bir yürüyüş türüdür. Bilinen yürüyüş (known walk) ise bir cevaba ulaşmak için gezilecek kenar ve düğümler hakkında önceden bilgi sahibi olmayı ifade eden bir modeldir. Patika bulma algoritmalarında gezilecek kenar ve düğümler bilinmesine rağmen, adım sayısı bilinmez. Ancak bilinen yürüyüş problemlerinde, gezilecek kenar ve düğümler ile birlikte adım sayısı da bilinmek zorundadır. Diğer biri ifadeyle bilinen yürüyüşte, patikanın optimize edilmesine olanak tanıyan patikayla birlikte aynı patikanın kaç kez kullanıldığı (ya da adım sayısı bilgisi) de bilinir (Bechberger ve Perryman, 2020: 175).

Örneğin restoran tavsiyesi yapan bir grafta, “Belirli bir mutfağa sahip yakınımdaki hangi restoran en yüksek puanı alıyor?” sorusunun cevabı için başlangıç düğümünden bitiş düğümüne varmak amacıyla, gezinilmesi gereken adımlar tam olarak bilinmelidir. Çünkü döndürülen düğüm “restaurant”, restoranın puanı “değerlendirme puanı” (review) ve belirli bir mutfak kültürüne göre filtrelenen “mutfak” (cuisine) ögeleri kullanılmak zorundadır (Şekil 3.8).




Şekil 3.8. Bilinen yürüyüş için restoran tavsiyesi grafı, (Bechberger ve Perryman, 2020: 174)

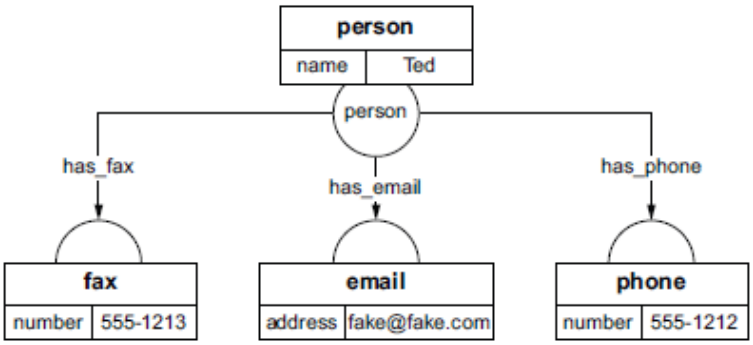
Her ögenin diğeriyle tekil bir ilişkisi olduğundan, ögeler arasında gezinmek için bilinen belirli bir patika vardır. Sonucu elde etmek için düğümler arasında yapılan gezinme sayısı bilinmelidir. Dolayısıyla restoranla bağlantılı düğümler çağrıldığında, hizmet (serves) ve puanlama (are about) kenarlarında gezinmek gerektiği bilinmektedir (Bkz. Şekil 3.8).

Benzer şekilde "En yüksek puanlanan yakınımdaki 10 restoran hangileridir?" sorusunda, düğümler arasındaki ilişkiler ve düğümler açık bir şekilde tanımlanmıştır. Çünkü döndürülen düğüm "restoran" ve restoranın puanı "değerlendirme puanı" (review) düğümü bilinmektedir. Restoran ve değerlendirme puanı arasında tek bir ilişki olduğu için patika da tanımlanmıştır. Sonucu elde etmek için düğümler arasındaki gezinme sayısı da bilindiğinden, bu problem de bilinen yürüyüş olarak ifade edilebilir. Çünkü her restoranın aldığı en yüksek puanı hesaplamak için gerekli olan "değerlendirme puanı" (review) düğümüne varmak amacıyla, "puanlama" (are about) ilişkisi üzerinde tek iterasyon yeterlidir (Bkz. Şekil 3.8).

## 3.7. Grup Etiketi

Grup etiketi (generic label), ögelerin benzerliklerine göre kategorize edilerek düğüm ya da kenarların etiketlenmesidir. Ögelerin gruplandırılması suretiyle daha az sayıda nesne türü üzerinde gezinmek ve onları modellemek basitleşir. Ancak çok sayıda grup etiketi kullanmak, gruplandırmanın getirdiği avantajları ortadan kaldırarak gezinme performansını düşürür. Hem gezinmeleri kolaylaştırmaya, hem de nesnelerin en iyi temsilini sağlamaya yeter sayıda grup etiketi kullanılmalıdır (Bechberger ve Perryman, 2020: 177). O halde

optimum grup etiketi sayısının belirlenmesi gerekir. Grup etiketlerinin işleyişini incelemek amacıyla kişi (person), email, telefon ve fax düğümlerinden oluşan temsili graftan yararlanılabilir (Şekil 3.9).




Şekil 3.9. Grup etiketlerinin işleyişi, (Bechberger ve Perryman, 2020: 178)

Gezinmeleri optimize etmek ve basitleştirmek için nesnelerin kategorize edilmesi, grup etiketi kullanmanın temel sebeplerinden biridir. Bu çerçevede grup etiketlerini incelemek için aşağıda üç farklı gezinme senaryosu sunulmuştur.

- "Ted'in telefon numarası nedir?" sorusunun cevabına ilişkin aşağıdaki sorguda, sadece person düğümüne filtreleme yapılarak (has_phone isimli) tek bir kenarda gezinilir.

```
g.V().has('person', 'name', 'Ted').
out('has_phone').values('number')
```

- "Ted'in tüm iletişim bilgileri (telefonu, e-posta adresi, fax no.su) nelerdir?" sorusunun cevabına ilişkin sorguda, sonuçları birleştiren bir "union()" adımı kullanıldığından, (has_phone, has_email ve has_fax isimli) üç kenarda gezinilir. Bu yüzden arzulanan performanslı bir gezinme değildir.

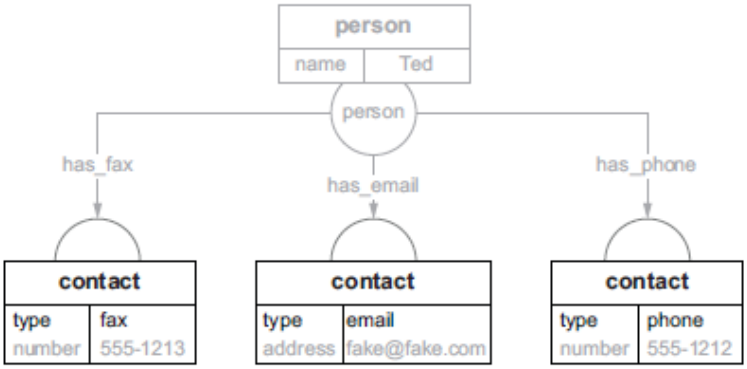
```
g.V().has('person', 'name', 'Ted').
union(
out('has_phone' ).values('number'),
out('has_email').values('address'),
out('has_fax').values('number')
)
```

- “Sistemdeki insanların tüm iletişim bilgileri (telefonu, e-posta adresi, fax no.su) nelerdir?” sorusunun cevabına ilişkin sorguda, bir önceki örneğe benzer şekilde sonuçları birleştiren bir “union()” adımı kullanıldığından, (has_phone, has_email ve has_fax isimli) üç kenarda gezinilir. Bu yüzden bu sorgu da arzulanan bir performanslı gezinme değildir.

```
g.V().
union(
out('has_phone').values('number'),
out('has_email').values('address'),
out('has_fax').values('number' )
)
```

Yukarıdaki ikinci ve ve üçüncü sorguda yer alan “union()” ifadesi, “union(gezinme, gezinme, ...)” şeklinde gezinmeleri parametre olarak alır ve her bir iletişim bilgisi için gezinmeyi ayrı ayrı yürütüp sonuçları birleştirir. “union()” ifadesi, büyük ölçekli graflarda önemli bir performans kaybına yol açar. Çünkü (has_phone, has_email ve has_fax gibi) her bir kategoride “union()” adımların yürütülmesi (running) amacıyla, mevcut gezinmenin kopyalanması gerekir. Diğer bir ifadeyle son iki gezinme, sürecin devam edebilmesi için gezinmeyle ilgili hesaplamaların üç kez kopyalanmasını / yapılmasını gerektirir.

Her biri aynı mantıksal yapıya temsil eden email, phone ve fax etiketleri, “contact” isimli bir grup etiketinde birleştirilebilir. Bu yöntemin maliyeti (email, phone ve fax gibi) iletişim türü bilgisinden vazgeçmektir. Spesifik bir etiketten grup etiketine geçmenin söz konusu maliyeti, “contact” düğümüne eklenen “type” özelliğiyle kolaylıkla telafi edilebilir (Şekil 3.10).




Şekil 3.10. Grafın tüm düğümlerine "contact" grup etiketinin eklenmesi, (Bechberger ve Perryman, 2020: 179)

Grafın tüm düğümlerine eklenen "contact" grup etiketi sonrası, yukarıdaki üç senaryonun sorgularında görülen değişimler aşağıda değerlendirilmiştir.

- "Ted'in telefon numarası nedir?" sorusunun cevabına ilişkin aşağıdaki sorguda, bir önceki sorguya göre herhangi bir değişiklik yoktur.

```
g.V().has('person', 'name', 'Ted').
out('has_phone').values('number')
```

- "Ted'in tüm iletişim bilgileri (telefonu, e-posta adresi, fax no.su) nelerdir?" sorusunun cevabına ilişkin aşağıdaki sorguda, bir önceki sorguya göre herhangi bir değişiklik yoktur.

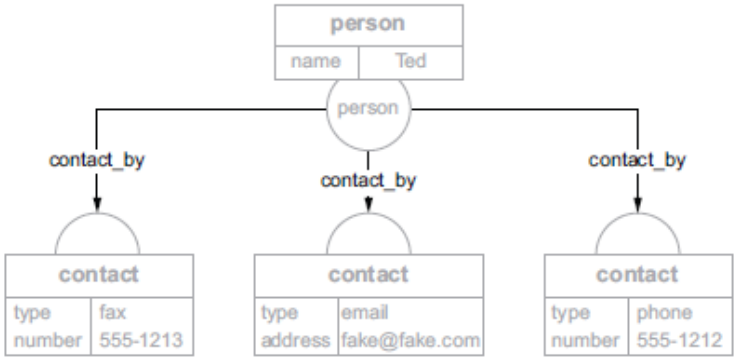
```
g.V().has('person', 'name', 'Ted').
union(
out('has_phone' ).values('number'),
out('has_email').values('address'),
out('has_fax').values('number')
)
```

- "Sistemdeki insanların tüm iletişim bilgileri (telefonu, e-posta adresi, fax no.su) nelerdir?" sorusunun cevabına ilişkin aşağıdaki sorgu, "union()" adımını kullanmadığından bir önceki sorguya göre farklıdır. Bu yüzden bu sorgu arzulanan performanslı bir gezinme örneğidir.

```
g.V().
```

```
hasLabel('contact').values('number', 'address', 'type')
```

Yukarıda üç düğüme uygulanan grup etiketi, “contact_by” isimli bir etiket kullanılarak düğümlere ilişkin kenarlara da uygulanmak suretiyle ikinci sorgunun performansı da geliştirilebilir (Şekil 3.11).




Şekil 3.11. Grafın tüm kenarlarına “contact_by” grup etiketinin eklenmesi, (Bechberger ve Perryman, 2020: 180)

Grafın tüm kenarlarına eklenen “contact_by” grup etiketi sonrası, yukarıdaki üç senaryonun sorgularında görülen değişimler aşağıda değerlendirilmiştir.

- “Ted’in telefon numarası nedir?” sorusunun cevabına ilişkin aşağıdaki sorguda, bir önceki sorguya göre “has('contact', 'type', 'phone')” şeklinde ilave bir filtre yer almaktadır. Söz konusu filtre kenarlara uygulanan grup etiketinin maliyetini telafi etmektedir.

```
g.V().has('person', 'name', 'Ted').
out('contact_by').
has('contact', 'type', 'phone').values('number', 'type')
```

- “Ted’in tüm iletişim bilgileri (telefonu, e-posta adresi, fax no.su) nelerdir?” sorusunun cevabına ilişkin aşağıdaki sorgu, gezinmenin tek bir kenarda gerçekleşmesi anlamında bir önceki sorguya göre farklıdır. Bu yüzden bu sorgu da arzulanan performanslı bir gezinme örneğine dönüştürülmüştür.

```
g.V().has('person', 'name', 'Ted').
out('contact_by').values('number', 'address', 'type'
```

- “Sistemdeki insanların tüm iletişim bilgileri (telefonu, e-posta adresi, fax no.su) nelerdir?” sorusunun cevabına ilişkin aşağıdaki sorguda, bir önceki sorguya göre herhangi bir değişiklik yoktur.

```
g.V().
hasLabel('contact').values('number', 'address', 'type')
```

Yukarıdaki süreçte de detaylı olarak incelendiği üzere, grup etiketlerine dayalı olarak gerçekleştirilen veri modelindeki değişimler, bazı gezinmelerde pozitif bazılarında ise negatif etkilere yol açabilmektedir. Bu yüzden veri modeli optimizasyonu, hayattaki birçok şey gibi değiş-tokuş (tradeoff) içerir. Bunlar arasındaki dengeyi ve değiş-tokuşu anlamak, en yaygın kullanılan gezinme örüntülerini optimize eden bir veri modelinin oluşturulmasında önemli olduğu ifade edilebilir (Bechberger ve Perryman, 2020: 181).

### 3.8. Denormalizasyon

Graflarda denormalizasyon (denormalization), okuma performansını arttırmak amacıyla, verinin yazma anında graf üzerinde farklı konumlara kopyalanmasıdır. Bu çerçevede denormalizasyon, bir özelliğin okunması için graf boyunca gezinmenin maliyetinden kaçınarak, okuma yoğun belirli işlemlere yardımcı olan bir yöntemdir. Bir özellik değerinin birden çok veri kopyasının tutulmasını ifade eden denormalizasyonun, kopyalama sürecinden kaynaklanan üç maliyeti vardır:

- Verinin birden fazla konuma yazılmasından dolayı verinin hacmi artar. Bu yüzden denormalizasyon özelikle büyük ölçekli verilerde disk kullanımını arttırır.
- Benzer şekilde verinin çok sayıda konuma yazılmasından dolayı, her değişiklik yapıldığında tüm konumların eş anlı olarak güncellenmesi gerekir. Çünkü konumların güncellenmemesi durumunda, farklı gezinmeler doğal olarak farklı sonuçlar getirir.
- Tüm konumların senkronize olarak güncellenmesi ihtiyacı daha fazla yazma operasyonu gerektirdiğinden dolayı, bu süreç “yazmanın büyümesi” (write amplification) olarak da adlandırılır.

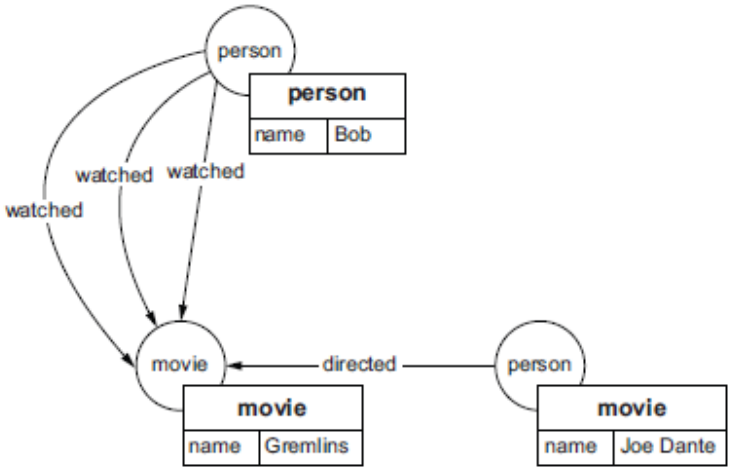
Yukarıdaki maliyetlere rağmen normalize bir veri modelinde verinin çağrılması (okunması) yeterince hızlı değilse, “okuma” performansını arttırmak için denormalize veri modeli

kullanılabilir. Zayıf okuma performansına sahip veri tabanında, bilginin çağrılabilmesi genellikle çok sayıda operasyon gerektirir. Söz konusu durum, özellikle dağıtık sistemlerde ilave ağ, hafıza ve disk erişimine olan ihtiyacı arttırdığından verimsiz süreçlere yol açar (Bechberger ve Perryman, 2020: 183). Bu yüzden graflarda denormalizasyon, başlangıç düğümünden bitiş düğümüne varmak için gerekli gezinme uzunluğunu kısalttığından, önemli bir performans optimizasyonu tekniğidir.

Çok sayıda denormalizasyon tekniği vardır. Bununla birlikte en yaygın kullanılan iki denormalizasyon yöntemi önceden hesaplanmış alanlar (precalculated fields) ve verinin çoğaltılması (duplicated data) teknikleridir.

### 3.8.1. Önceden hesaplanmış alanlar yöntemi

Denormalizasyon tekniklerinden biri olan önceden hesaplanmış alanlar (precalculated fields), okuma zamanında verinin hızlı bir şekilde çağrılmasını sağlamak için yazma zamanında hesaplamaların yapılarak, elde edilen sonucun düğümün ya da kenarın özelliğinde saklanmasını ifade eder. Eğer bir gezinme düğümlerin güncellenmesinden ziyade daha çok toplam, ortalama, sayma gibi toplulaştırma işlemlerini yapıyorsa, bir düğüme ya da kenara önceden hesaplanmış alanlar eklemek işlem performansını önemli ölçüde iyileştirecektir (Bechberger ve Perryman, 2020: 184).

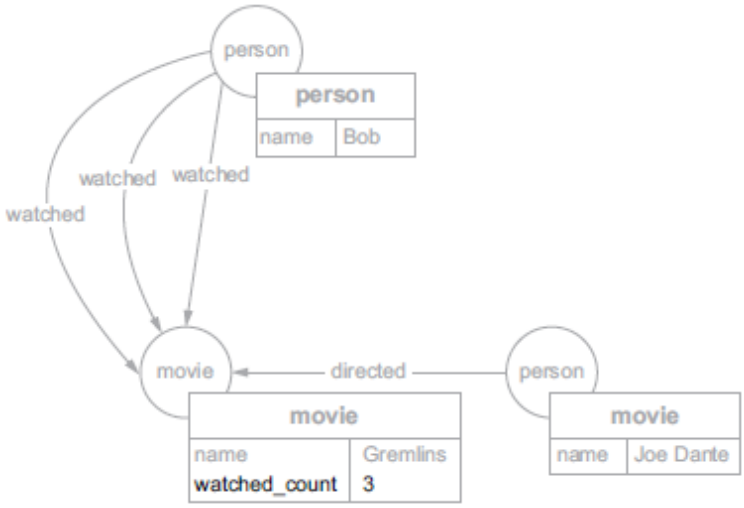



Şekil 3.12. Önceden hesaplanmış alanlar yöntemi,
(Bechberger ve Perryman, 2020: 184)

Önceden hesaplanmış alanların çalışma ilkelerini incelemek için Şekil 3.12'de sunulan temsili graftan yararlanılabilir. Bu çerçevede "Kaç kişi Gremlins isimli filmi izledi?" sorusunun cevabına ilişkin sorgu aşağıdadır.

```
g.V().has('movie', 'name', 'Gremlins').
bothE('watched').
count()
```

Bu gezinme "watched" isimli kenarlarının sayısını döndürerek "Gremlins" isimli filmi izleyen kişi sayısını hesaplar (Bkz. Şekil 3.12). Ancak "watched" kenarlarının sayısı, hesaplama süresinin doğrusal bir fonksiyonudur, yani kenar sayısı arttıkça hesaplama süresi de artar. Popüler filmler, popüler olmayan filmlerden daha fazla izlendiğinden, popüler film düğümlerinin hesaplanması popüler olmayan film düğümlerinin hesaplanması süresinden daima daha fazla olur. Bu problem, "movie" düğümü üzerine "watched" kenarlarının sayısını içeren "watched_count" özelliği yerleştirilerek çözülebilir (Şekil 3.13).



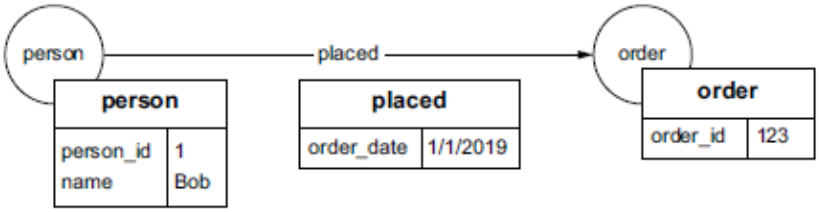

Şekil 3.13. "watch_count" özelliğinin eklenmesi, (Bechberger ve Perryman, 2020: 185)

Bir "watched" kenarı eklendiğinde, güncellendiğinde ya da silindiğinde, "movie" düğümünün "watched_count" özelliğinde yer alan bilgi de değiştirilerek güncellenir.

"Gremlins" isimli filmi izleyen kişi sayısı bilgisine ulaşmak için herhangi bir ilave hesaplama yapmadan "movie" düğümünde yer alan "watched_count" özelliği içinde tutulan değerin okunması yeterli olacaktır. İşlem performansı da "watched" kenarı sayısından bağımsız olacağından zamanla (kenar sayısı arttıkça) kötüleşmez.

### 3.8.2. Veri çoğaltma yöntemi

Denormalizasyon tekniklerinden bir diğeri veri çoğaltma (duplicate data), bir düğüm ya da kenar özelliğinin diğer bir düğüm ya da kenar özelliğine kopyalanmasını ifade eder. Özelliklerin graf üzerinde birden fazla konuma kopyalanması, saklanan verinin yazma zamanındaki senkronizasyon maliyeti karşılığında, çoklu patikalarda gezinmeyi optimize ederek okuma zamanındaki performansının iyileştirilmesini sağlar. Eğer düğümler ya da kenarlar üzerinde çok sayıda gezinmeyi gerektiren türde bir sorgu söz konusu ise erişilmek istenen veri, gezinme sürecinin önceki adımlarına kopyalanarak gezinme performansı iyileştirilebilir (Bechberger ve Perryman, 2020: 186).




Şekil 3.14. Veri çoğaltma yöntemiyle, (Bechberger ve Perryman, 2020: 186)

Şekil 3.14'e göre "123 siparişini hangi tarihte verdim?" sorusunun cevabına ilişkin sorgu aşağıdadır.

```
g.V().has('order', 'order_id', '123').
outE('placed').
values('order_date')
```

Yukarıdaki sorguya göre ilk olarak, "order_id" özelliğinin 123 değerine ulaşmak için "order" düğümü bulunur. Sonra da "placed" kenarında gezinilerek "order_date" özelliğinde saklanan değere erişilir. Bu türden bir sorgu sık sık kullanılıyorsa, ilave yükler potansiyel bir darboğaz oluşturur.


Şekil 3.15. Kenarda tutulan bilginin düğüme kopyalanması,
(Bechberger ve Perryman, 2020: 186)

Şekil 3.15'te gösterildiği gibi "placed" kenarının "order_date" özelliğinde saklanan değerin kopyası, "order" düğümünün özelliği içine yerleştirilerek, sorgunun tek adımda sonuçlandırılması sağlanır. Böylece gereksiz düğümlerde ya da kenarlarda gezinmeye ihtiyaç kalmaz.

### 3.9. Alt Graflar

Alt graf ise (subgraph) tüm düğümleri ve kenarları global ya da daha büyük bir grafın alt kümesinden oluşan daha küçük bir grafı ifade eder. Kişiselleştirme (personalization), en ilgili içeriği elde etmek için verilerdeki bağlantılara göre verinin filtrelenmesi işlemi olarak tanımlanabilir.

Alt graflar örnek bir senaryo üzerinden incelenebilir. Nancy ve Sam, farklı semtlerde yaşadıkları için farklı arkadaş grupları vardır ve birbirleriyle tanışmamışlardır. Ancak her ikisi de kendi arkadaş gruplarının yüksek puanladıkları aynı restorana gitmek istemektedirler. Bu çerçevede "Arkadaşlarımın puanladığı restoranlara göre bana en yakın restoran hangisidir" sorusunun cevabı için grafın tümü yerine, Nancy ve Sam'a ilişkin kısımlarını temsil eden (kişiselleştirilmiş) iki farklı alt grafta işlem yapmak sorgulama performansını önemli ölçüde arttırır (Bechberger ve Perryman, 2020: 237-238).

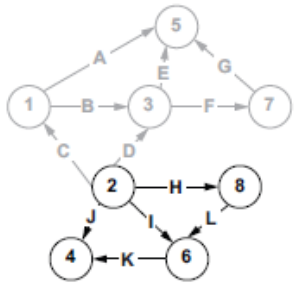
Alt graf çıkartmanın / türetmenin ilkelerini incelemek amacıyla, Şekil 3.16'da Josh ve onun arkadaşlarıyla ilgili alt grafın çağrılmasının istendiği varsayılmıştır.


Şekil 3.16. Alt graf çıkartmak, (Bechberger ve Perryman, 2020: 239)

Söz konusu alt grafı global graftan elde etmek amacıyla, kenarları ve düğümleri tanımlayan bir gezinme geliştirilmelidir. Birinin (Josh'ın) arkadaşlarını bir alt grafın parçası olarak belirlemek için "düğüm tetikleme" (vertex-induced) ve "kenar tetikleme" (edge-induced) şeklinde iki yöntem vardır.

### 3.9.1. Düğüm tetikleme

Düğüm tetiklemede (vertex-induced) alt graf, bir dizi düğüm ve paylaşılan herhangi bir kenar tarafından belirlenir. Şekil 3.17'de yalnızca çift sayılarla temsil edilen, yani (H, I, J, K ve L) ortak kenarlara sahip düğümler kullanılarak oluşturulan bir alt graf gösterilmektedir.

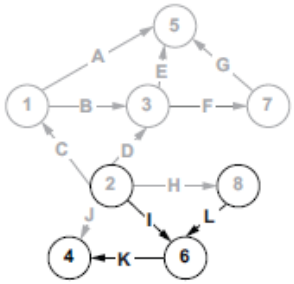



Şekil 3.17. Düğüm tetikleme yöntemi, (Bechberger ve Perryman, 2020: 240)

Şekil 3.17'de, 6 no.lu düğüm esas alınarak düğüm tetikleme yöntemiyle, 6 no.lu grafın komşu düğüm ve kenarları ile komşu düğümleri arasındaki kenarlarından oluşan bir alt graf türetilmiştir.

### 3.9.2. Kenar tetikleme

Kenar tetiklemede (edge-induced) alt graf, bir dizi kenar ve olay düğümü tarafından belirlenir. Şekil 3.18'de 6 no.lu olay düğümüyle bağlantılı (I, K ve L) kenarlara göre oluşturulan (2, 4, 6 ve 8 no.lu düğümleri içeren) bir alt graf gösterilmektedir.




Şekil 3.18. Kenar tetikleme yöntemi, (Bechberger ve Perryman, 2020: 240)

Şekil 3.18'de, 6 no.lu düğüm esas alınarak kenar tetikleme yöntemiyle, 6 no.lu grafın sadece komşu düğüm ve kenarlarından oluşan bir alt graf türetilmiştir.

Düğüm tetikleme yöntemiyle elde edilen bir alt graf, paylaşılan tüm kenarları içermesine rağmen, kenar tetikleme yöntemiyle elde edilen bir alt graf ise sadece olay düğümüyle ilgili kenarları içerir. Ancak bu iki yöntem her zaman farklı sonuçlar üretmez (Bechberger ve Perryman, 2020: 240).

Kenar tetikleme yöntemiyle alt graf oluşturmaya ilişkin Gremlin sorgusu aşağıda gösterilmektedir.

```
g.V().has('person','person_id',2).
bothE('friends').
subgraph('sg').
cap('sg').
next()
==>tinkergraph[vertices:4 edges:3]
```

Birinci ve ikinci satırdaki ifadeler, "person" düğümünün bulunarak her iki yönde "friends" kenarları boyunca gezinmeyi temsil eder. Üçüncü ifadeyle gezinilen kenarlardan "sg" key değerine sahip bir alt graf oluşturulur. Dördüncü ifadeyle "sg" anahtar değerine sahip alt graf içinde kenarlar ve düğümler bir araya getirilir. Beşinci ifade ise sonucu döndürür. Son ifade ise 4 düğüm ve 3 kenar kullanılarak oluşturulan alt grafa ilişkin döndürülen değerdir.

Gremlin'de alt graflarda gezinmek için aşağıdaki kodun çalıştırılması gerekir.

```
sg = subgraph.traversal()
```

"sg" ifadesi sadece bir değişkendir. Alt grafta gezinmek için aşağıda yer alan örnek kodlar incelenebilir.

```
sg.V().has('person','person_id',2).valueMap()
==>{person_id=[2], last_name=[Perry], first_name=[Josh]}
```

Yukarıdaki örnekte "sg" alt grafında, id no.su 2 olan düğüm ziyaret edilmiş ve düğümün özellik anahtarı ve değeri görüntülenmiştir. Söz konusu düğümün tüm bağlantılarını bulmak ve onların özellik anahtarı ile değerini görüntülemek için aşağıdaki sorgudan yararlanılabilir.

```
sg.V().has('person','person_id',2).both().valueMap()
==>{person_id=[3], last_name=[Erin], first_name=[Hank]}
==>{person_id=[1], last_name=[Bech], first_name=[Dave]}
==>{person_id=[4], last_name=[Wilson], first_name=[Ted]}
```

# 4. GELİŞMİŞ GRAF İŞLEME YÖNTEMLERİ

Graf veri tabanlarında gelişmiş veri işleme yöntemleri temel olarak beş kategoride sınıflandırılabilir.

*Graf algoritmaları (graph algorithms)*: Graf algoritmaları, graf veri işleme için özel olarak tasarlanmış matematiksel algoritmalardır. Yaygın olarak kullanılan graf algoritmalarından bazıları, BFS (breadth-first search), DFS (depth-first dearch), pagerank, en kısa patika (shortest path), topluluk tespiti (community detection) ve merkezilik (centrality) algoritmalarını içerir. Bu algoritmalar, düğümler arasındaki en kısa yolu bulma, graf içindeki toplulukları veya kümeleri belirleme, önemli düğümleri veya kenarları belirleme gibi çeşitli graf analizi görevlerini gerçekleştirmek için kullanılır.

*Paralel işleme (parallel processing)*: Paralel işleme teknikleri, grafı daha küçük alt graflara (subgraph) bölerek ve bunları birden fazla işlemci veya bilgisayarda aynı anda işleyerek graf veri işlemeyi hızlandırmak için kullanılır. Paralel işleme, paylaşılan bellek paralelliği (shared-memory parallelism), dağıtık bellek paralelliği (distributed-memory parallelism), GPU hızlandırma gibi teknikler kullanılarak gerçekleştirilebilir. Paralel işleme, graf algoritmalarının işlem süresini önemli ölçüde azaltabilir ve özellikle büyük ölçekli graf veri tabanları için etkilidir.

*Dağıtık işleme (distributed processing):* Dağıtık işleme, grafın daha küçük alt graflara bölünmesini ve bunların birden çok makinede işlenmesini içerir. Dağıtık işleme, büyük ölçekli veri işlemeyi dağıtılmış bir şekilde işlemek için tasarlanmış Apache Hadoop veya Apache Spark gibi dağıtık bilgi işlem kütüphaneleri (framework) kullanılarak gerçekleştirilebilir. Dağıtık işleme, tek bir makinede işlenemeyen büyük ölçekli graflar için son derece kullanışlıdır.

*İndeksleme ve ön belleğe alma (indexing and caching):* Graf veri tabanları, sık erişilen verileri önceden hesaplamak ve depolamak için genellikle indeksleme ve ön belleğe alma tekniklerini kullanır. Sık erişilen veriler önbellekten veya dizinden daha hızlı alınabileceğinden, dizin oluşturma ve ön belleğe alma, sorgu performansını artırmaya ve işlem sürelerini azaltmaya yardımcı olabilir. İndeksleme ve ön belleğe alma, B+ ağaçları

(B+ trees), hash indeksleri (hash indexes) ve bloom filtreleri (bloom filters) gibi çeşitli teknikler kullanılarak gerçekleştirilebilir.

*Donanım hızlandırma (hardware acceleration):* Donanım hızlandırma, graf veri işlemeyi hızlandırmak için GPU'lar (graphics processing unit) veya FPGA'lar (field programmable gate arrays) gibi özel donanımların kullanılmasını içerir. Donanım hızlandırıcılar, graf veri işleme görevlerini geleneksel işlemcilerden çok daha hızlı gerçekleştirebilir ve özellikle graf gezinmesi (graph traversal) ve topluluk algılama (community detection) gibi görevler için etkilidir. Donanım hızlandırma yöntemiyle en iyi performansı elde etmek için paralel işleme veya dağıtık işleme gibi diğer işleme yöntemleriyle birlikte kullanılabilir.

Etkin graf işleme yöntemleri, büyük ölçekli graf veri tabanları için gereklidir. Yöntem seçimi (ya da kombini) grafın boyutu ve karmaşıklığı, gerçekleştirilen görevler, mevcut donanım ve yazılım kaynakları gibi çeşitli faktörlere bağlı olacaktır. Büyük ölçekli graf veri tabanları için en uygun performansı elde etmek için bu yöntemlerin bir kombinasyonu da kullanılabilir.

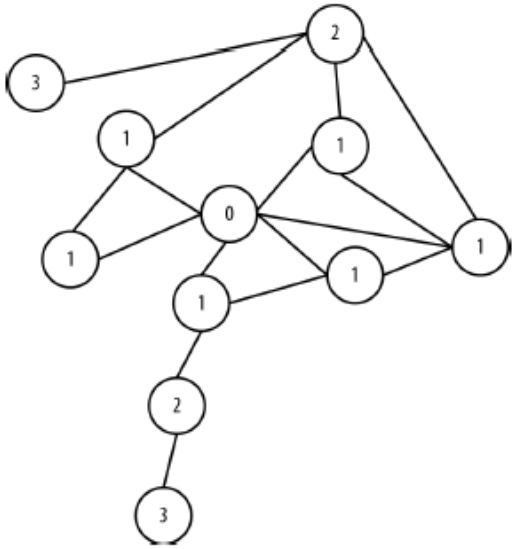
Bu çalışmanın takip eden kısmında gelişmiş graf işleme yöntemlerinden yaygın kullanılan graf algoritmaları üzerinde durulacaktır. Bu çerçevede graf algoritmaları, büyük ölçekli graflarda performansları da göz önüne alınmak suretiyle, üç temel kategori altında incelenecektir: arama ve patika bulma (graph search and pathfinding), merkezilik (centrality) ve topluluk algılama (community detection) algoritmaları.

### 4.1. Arama ve Patika Bulma Algoritmaları

Arama (search) algoritmaları, grafın genel olarak keşfedilmesi için bir tarama yöntemidir. Genişlik öncelikli arama (breadth-first search, BFS) ve derinlik öncelikli arama (depth-first search, DFS) bir grafın tümünü haritalandıran ve hedefe yönelik patika bulma (pathfinding) algoritmalarla birlikte kullanılan en temel tarama algoritmalarıdır. Patika bulma (pathfinding) algoritmaları ise graf arama algoritmalarının üzerine inşa edilir. Bir başlangıç düğümüyle bitiş / hedefe düğümü arasında bulunan kenarlarda dolaşarak düğümler arasındaki yolları keşfeder.

### 4.1.1. Genişlik öncelikli arama

Genişlik öncelikli arama (breadth-first search, BFS) algoritmasında gezinmeler, öncelikli olarak grafın (derinliği yerine) genişliği boyunca gerçekleşir. Diğer bir ifadeyle graf, başlangıç düğümünden itibaren önce birinci derinlikteki düğümler, ardından ikinci derinlikteki düğümler, ardından üçüncü derinlikteki düğümler şeklinde katman katman ziyaret edilir. Tüm graf taranana kadar işlemler aynı şekilde devam eder (Robinson ve diğerleri, 2015: 172). Bu süreç, 0 (orijin için) no.lu düğümden başlanarak ve her seferinde bir katmandan dışarı (komşu düğümlere ve komşu düğümlerin komşu düğümlerine) doğru ilerlenerek görselleştirilmiştir (Şekil 4.1).




Şekil 4.1. BFS algoritması, (Robinson ve diğerleri, 2015: 172)

BFS algoritması, bir düğümün birinci dereceden tüm komşularını ziyaret etmeden önce, sonraki derinlikte yer alan komşu düğümlerini keşfetmeye çalışmaz. Bu yönüyle BFS, daha çok hedefe yönelik algoritmaların temeli olarak yaygın şekilde kullanılır. Örneğin, en kısa patika (shortest path), yakınlık merkeziliği (closeness centrality) gibi yöntemler BFS algoritmasını kullanır (Needham ve Hodler, 2019: 44).

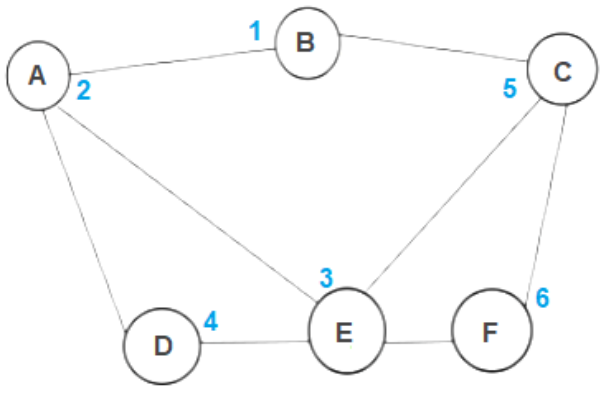
### 4.1.2. Derinlik öncelikli arama

Derinlik öncelikli arama (depth-first search, DFS), bir grafın düğümlerini ziyaret etmenin sistematik bir yöntemidir. DFS her adımda en son ziyaret edilen komşuya sahip bir düğümü öncelik olarak seçer. Örneğin v düğümünde bulunulduğu varsayımında, v'nin henüz ziyaret

edilmemiş bir düğümünün ziyaret edilmesini ifade eder. Böyle bir düğüm yoksa, v'den hemen önce ziyaret edilen düğüme geri dönülerek arama, grafın her düğümü ziyaret edilene kadar tekrarlanır.

DFS algoritması girdi olarak bir G grafı alır ve önceki alt grafını bir derinlik öncelikli orman (depth-first forest) biçiminde çıkarır. Ek olarak, her düğüme iki zaman damgası atar: keşif ve bitiş zamanı. Algoritma, henüz keşfedilmediklerini belirtmek için tüm düğümleri (n) "beyaz" olarak başlatır. Henüz keşfedilmemiş beyaz işaretli komşu düğümler v ile temsil edilir. Ayrıca her düğümün ebeveynini “null” olarak ayarlar. Algoritma, graftan bir u düğümü seçerek, düğümün artık keşfedildiğini (ancak tamamlanmadığını) belirtmek için rengini "gri" olarak ayarlar ve ona keşif süresi 0 değerini atayarak başlar. Algoritmanın esas bölümünü temsil eden yinelemeli kısım çalışarak sırayla tüm v komşu düğümlerine uygun keşif zamanı değerlerini atar. v düğümünün selefinde beyaz yoksa, v düğümü siyah olur ve uygun bitirme zamanı atanarak algoritma v'nin atasının keşfine geri döner. Eğer u'nun tüm selefleri siyah ise u siyah olur. Kalan beyaz düğümlerden yeni bir “kaynak” düğüm seçilir ve işleme devam edilir. Eğer grafta başka beyaz düğüm yoksa algoritma sonlanır.

DFS'nin başlatma kısmı O(n) zaman karmaşıklığına sahiptir. Çünkü her düğümün "beyaz" olarak işaretlenmesi için bir kez ziyaret edilmesi gerekir. Algoritmanın ana (yinelemeli) kısmı ise O(m) zaman karmaşıklığına sahiptir, Çünkü her düğümün komşu düğümlerinin incelenmesi sırasında her kenarın (iki kez) geçilmesi gerekir. Toplamda ise algoritmanın zaman karmaşıklığı O(m + n) olur (Papamanthou, 2004: 1; Cormen, Leiserson ve Rivest, Stein, 2001: 540).




Şekil 4.2. DFS algoritması

BFS algoritmasının işleyişi, Şekil 4.2'de yer alan temsili bir graf üzerinden açıklanacaktır. Bu amaçla başlangıç düğümünün B olduğu varsayımında, B'nin herhangi bir komşusunda, örneğin A düğümünde ilerlenir. A düğümün komşuları ise D ve E olduğu için onlardan herhangi birine, örneğin E düğümüne geçilir. E düğümünün komşuları ise C, D ve F olduğu için onlardan herhangi birine, örneğin D düğümüne geçilir. D düğümünün tüm komşuları (A ve E) daha önce gezildiği (işaretlendiği) için D düğümü üzerinden gezilecek (keşfedilmemiş) başka herhangi bir komşu düğüm kalmamıştır. Bu yüzden D düğümünün selefi E düğümüne geri dönülür.

E düğümünün komşularından D düğümü zaten keşfedildiği için diğer komşularından C ve F düğümlerinden herhangi birine, örneğin C düğümüne geçilir. C düğümünün komşularından B düğümü başlangıç düğümü (keşfedilmiş) olduğu için diğer komşusu F düğümüne geçilir. F düğümünün komşusu E daha önce gezildiği (işaretlendiği) için F düğümü üzerinden gezilecek (keşfedilmemiş) başka herhangi bir komşu düğüm kalmamıştır. Bu yüzden F düğümünün selefi C düğümüne geri dönülür. C düğümü üzerinden gezilecek (keşfedilmemiş) başka herhangi bir komşu düğüm kalmamıştır. Bu yüzden C düğümünün selefi E düğümüne geri dönülür. E düğümü üzerinden gezilecek (keşfedilmemiş) başka herhangi bir komşu düğüm kalmamıştır. Bu yüzden E düğümünün selefi A düğümüne geri dönülür. Son olarak A düğümü üzerinden gezilecek (keşfedilmemiş) başka herhangi bir komşu düğüm kalmamıştır. Bu yüzden A düğümünün selefi B düğümüne geri dönülerek algoritma sonlandırılır. Buna göre DFS algoritmasının gezinmesi, sırasıyla B – A – E – D – C – F düğümleri üzerinde gerçekleşir.

### 4.1.3. En kısa patika

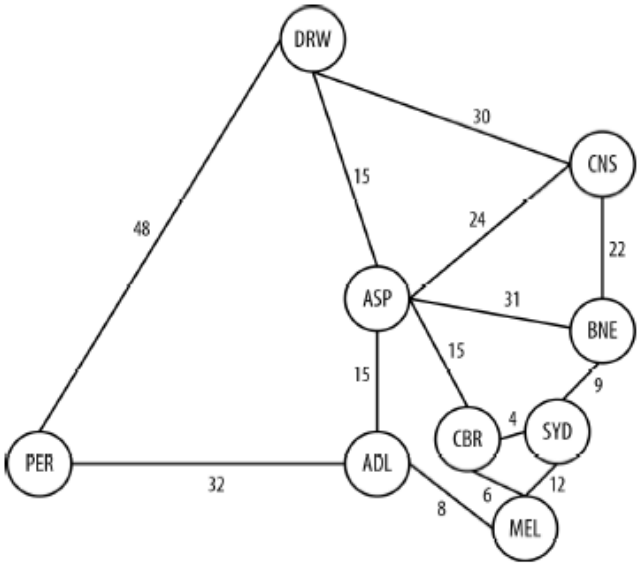
En kısa patika ( shortest path) algoritmaları, iki düğüm arasındaki en kısa (ağırlıklı) yolu hesaplar. Gerçek zamanlı çalıştığı için kullanıcı etkileşimleri ve dinamik iş akışları için kullanışlıdır. Patika bulma, 19. yüzyıla kadar uzanan bir geçmişe sahip olup klasik bir graf problemi olarak kabul edilir. 1950'lerin başında, en kısa patika bloke edilmişse en kısa ikinci patikayı bulma anlamında, alternatif rota belirleme algoritmaları önem kazandı. En bilineni ise 1956'da Edsger Dijkstra'nın algoritması oldu (Needham ve Hodler, 2019: 49-50). Daha sonra Dijkstra algoritması esas alınarak A* algoritması geliştirildi.

Dijkstra algoritması

Genişlik öncelikli arama algoritmalarından olan Dijkstra'nın algoritması bir graftaki düğümler arasındaki en kısa yolu bulmak için kullanılır. Dijkstra algoritmasının işleyişi beş adımda özetlenebilir.

- Başlangıç ve bitiş düğümleri seçilir ve başlangıç düğümü, 0 değeriyle çözülmüş düğümler kümesine eklenir.
- Genişlik öncelikli arama yaklaşımıyla, başlangıç düğümüne en yakın komşu düğümlerin uzaklığı hesaplanır.
- İkinci aşamada hesaplanan komşulardan en kısa yolu içeren düğüm çözülmüş düğümler kümesine eklenir.
- Genişlik öncelikli arama yaklaşımıyla, çözülmüş düğümler kümesinde yer alan düğümlerin (önceki aşamalarda çözülmüş düğümler hariç) en yakın komşuları ziyaret edilir ve başlangıç düğümünden bu yeni komşulara karşı uzunluklar hesaplanır.
- Hedef düğüm çözülene kadar üçüncü ve dördüncü adımlar tekrarlanır.

Dijkstra genellikle navigasyon örneğinde olduğu gibi gerçek dünyadaki en kısa yolları bulmak için kullanılır (Robinson ve diğerleri, 2015: 173).




Şekil 4.3. Dijkstra algoritması, (Robinson ve diğerleri, 2015: 174)

Şekil 4.3'te her düğüm bir şehri ve kenarlar ise şehirler arasındaki uzaklığı ifade etmektedir. Dolayısıyla kenarların uzaklık değerleriyle ağırlıklandırıldığı bir grafı temsil eder. SYD düğümünün başlangıç düğümü (hareket edilecek şehir), PER düğümünün ise bitiş düğümü (gidilecek şehir) olduğu kabul edilmiştir. SYD şehrinden PER şehrine en kısa rotanın elde edilmesi amaçlanmaktadır.

*1. Adım:* Başlangıç düğümü olan SYD'ye giden en kısa yol (zaten SYD düğümünde bulunulduğu için) 0 saat olarak çözüm kümesine eklenir. SYD'nin en yakın komşusu 4 km uzaklığa sahip CBR düğümü olduğu için CBR çözüm kümesine eklenir.

*2. Adım:* CBR düğümünün komşuları ASP ve MEL'in SYD düğümüne uzaklığı sırasıyla, 19 ve 10 km; SYD düğümünün kendi komşusu BNE ise 9 km uzaklığa sahip olduğu için BNE çözüm kümesine eklenir.

*3. Adım:* Çözüm kümesinde yer alan SYD, CBR ve BNE düğümlerinden, SYD başlangıç düğümüne en yakın komşuya sahip düğüm 10 km uzaklık ile (CBR'nin komşusu) MEL olduğu için MEL düğümü çözüm kümesine eklenir.

*4. Adım:* Çözüm kümesinde yer alan SYD, CBR, BNE ve MEL düğümlerinden, SYD başlangıç düğümüne en yakın komşuya sahip düğüm 18 km uzaklık ile (MEL'nin komşusu) ADL olduğu için ADL düğümü çözüm kümesine eklenir.

*5. Adım:* Çözüm kümesinde yer alan SYD, CBR, BNE, MEL ve ADL düğümlerinden, SYD başlangıç düğümüne en yakın komşuya sahip düğüm 19 km uzaklık ile (ADL'nin komşusu) ASP olduğu için ASP düğümü çözüm kümesine eklenir.

*6. Adım:* Çözüm kümesinde yer alan SYD, CBR, BNE, MEL, ADL ve ASP düğümlerinden, SYD başlangıç düğümüne en yakın komşuya sahip düğüm 31 km uzaklık ile (BNE'nin komşusu) CNS olduğu için CNS düğümü çözüm kümesine eklenir.

*7. Adım:* Çözüm kümesinde yer alan SYD, CBR, BNE, MEL, ADL, ASP ve CNS düğümlerinden, SYD başlangıç düğümüne en yakın komşuya sahip düğüm 34 km uzaklık ile (ASP'nin komşusu) DRW olduğu için DRW düğümü çözüm kümesine eklenir.

*8. Adım:* Son olarak, çözüm kümesinde yer alan SYD, CBR, BNE, MEL, ADL, ASP, CNS ve DRW düğümlerinden, SYD başlangıç düğümüne en yakın komşuya sahip düğüm 50 km uzaklık ile (ADL'nin komşusu) PER olduğu için PER düğümü çözüm kümesine eklenir.

ADL üzerinden PER düğümünün uzaklığı 50 km, DRW üzerinden PER düğümünün uzaklığı 82 km'den az olduğu için çözüm olarak ADL düğümü üzerinden rota tercih edilir. Diğer bir ifadeyle SYD düğümünden PER düğümüne en kısa mesafe, SYD-CBR-MEL-ADL-PER düğümlerine ilişkin patikadır (Robinson ve diğerleri, 2015: 174-181).

Yukarıdaki adımlar dikkatli bir şekilde incelendiğinde algoritma, 4. adımdan, yani ADL düğümünü çözüm kümesine ekledikten sonra, bitiş düğümü PER'e en yakın mesafenin ADL düğümü üzerinden olan yol olduğunu kaçırmaktadır. Çünkü Dijkstra algoritması başlangıç düğümü SYD'ye en yakın mesafeleri ölçerek SYD ile PER arasındaki en yakın mesafeyi hesaplamaya çalışmaktadır. Dolayısıyla algoritma sonraki adımda, çözüm kümesinde yer alan düğümlerden SYD başlangıç düğümüne en yakın komşuya yönelmektedir.

A* algoritması

Yukarıda da belirtildiği gibi Dijkstra, başlangıç düğümüne en yakın komşuları tercih ederek (hedef düğümü hesaba katmadan) hedefe doğru ilerleyen bir algoritma olduğu için fazladan gezinmeler yapabilmektedir. A* algoritması ise Dijkstra'nın başlangıç düğümüne en yakın komşu düğümleri tercih eden algoritmasıyla, hedefe daha yakın düğümleri tercih eden en iyi ilk aramanın (best-first search) özelliklerini birleştirir. Bu yönüyle Dijkstra algoritmasının geliştirilmiş bir versiyonunu temsil eden A* algoritması bir graftaki en kısa patikaları bulmak için en uygun çözümü sağlar.

A* algoritmasında patika bulma maliyeti iki aşamalıdır: g(n), "başlangıç düğümünden n düğümüne giden" yolun maliyetini ve h(n) ise "n düğümünden hedef düğüme giden" yolun tahmin maliyetini temsil eder. A* algoritması grafı yinelerken g(n) ve h(n)'yi dengeleyerek her yinelemede en düşük toplam maliyete (f(n) = g(n) + h(n)) sahip düğümü seçer (Robinson ve diğerleri, 2015: 181).

### 4.1.4. Rassal yürüyüş

Rassal yürüyüş (random walk) algoritması, bir grafta rassal bir yol üzerinde bir dizi düğüm sağlar. Algoritma gidilecek (hedef / bitiş) düğümü bilmesine rağmen, kendisini oraya ulaştıracak belirli bir yolu bilmediği için dolambaçlı bir yol izler. Eğer gezilecek düğümler arasında birer kenar varsa, algoritmanın iki düğüm arasında geçiş yapabileceği tek bir kenar olacağı için gezinme süreci rassal olmayacaktır. Ancak gezilecek düğümler arasında birden fazla kenar varsa, iki düğüm arasında yer alan alternatif kenarlardan biri seçileceği için gezinme süreci rassal olacaktır. Algoritma bir düğümde başlar ve giden veya gelen kenarlardan birini rassal bir şekilde seçerek komşu düğüme geçer. Daha sonra, hedef düğüme ulaşana kadar o düğümden itibaren iteratif bir şekilde aynı yöntemi takip eder.

Rassal yürüyüş algoritması, genellikle, rassal bir şekilde bağlantılı düğüm kümesi oluşturulması gerektiğinde, diğer algoritmaların veya veri hatlarının bir parçası olarak kullanılır (Needham ve Hodler, 2019: 73-74).

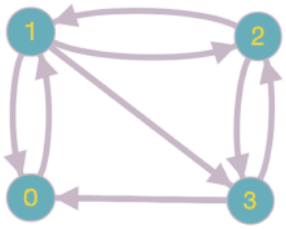
## 4.2. Merkezilik Algoritmaları

Merkezilik (centrality) algoritmaları, graf içindeki bir düğümün “önemini” (importance) belirlemek için kullanılır. Merkezilik, kaybı halinde en büyük zarara yol açacak bir bilgisayar ağındaki en kritik bileşeni, bir organizasyondaki en önemli kişiyi, sahtecilik tespitinde olduğu gibi graftaki aykırı (outlier) değerleri saptamada kullanılır.

### 4.2.1. Pagerank

Pagerank, en yaygın kullanılan graf algoritmalarından biridir ve bir graftaki düğümlerin önemini ölçmek için tasarlanmıştır. Pagerank algoritması, bir düğümün önemini, diğer önemli düğümlerden “gelen bağlantıların” sayısına bakarak hesaplar. Bu tanıma göre daha yüksek pagerank değerine sahip düğümlere bağlı düğümler, daha yüksek bir pagerank değerine sahip olma eğiliminde olacaktır. Bir düğümün pagerank'i yinelemeli olarak tanımlanır ve gelen bağlantılardan gelen tüm düğümlerin sayısına ve pagerank değerlerine bağlıdır. Pagerank, rassal bir yürüyüşün (random walker) her düğüme ulaşma olasılığını temsil eden bir olasılık dağılımı verir (Lim, Lee, Ganesh, Brown ve Sukumar, 2015).

Google'ın arama motorunun arkasındaki öncü yaklaşım olan pagerank, yönlü ve yönsüz grafların her ikisinde de başarılı bir şekilde uygulanabilir. Ancak pagerank dağılımının tam / analitik olarak hesaplanması zor olduğundan, tahmin edilmesi için çeşitli hesaplama yöntemlerinin kullanılır. Pagerank değerlerinin hesaplanması için temel olarak iki yaklaşım vardır: lineer cebir tekniklerine dayalı iterasyonlar ve Monte Carlo simülasyonları (Sarma, Molla, Pandurangan ve Upfa, 2015). Bu çalışma kapsamında pagerank değerlerinin lineer cebir yöntemleriyle hesaplanması üzerinde durulacaktır.




Şekil 4.4. Basit bir yönlü graf,
(Ahmed ve Thomo, 2021)

Şekil 4.4, dört düğümden oluşan basit bir yönlü grafı temsil etmektedir. 1 no.lu düğümün diğer üç düğüme, 2 no.lu düğümün 1 ve 3 no.lu düğümlere, 3 no.lu düğümün 2 ve 0 no.lu düğümlere ve son olarak 0 no.lu düğümün ise sadece 1 no.lu düğüme giden kenarları / bağlantıları vardır. Bir gezinmenin 1 no.lu düğümden kendisiyle bağlantılı (0, 2 ve 3 no.lu) düğümlerden herhangi birini ziyaret etme olasılığı 1/3 olur. Benzer şekilde 2 ve 3 no.lu düğümlerin 1/2, 0 no.lu düğümün ise 1/1 olur. Bağlantıların olasılık değerleri ağırlıklarıyla temsil edilebilir. Bu yüzden bir graf için (nxn) boyutlu bir M kare matrise karşılık gelen bir bitişiklik matrisi (adjacency matrix) oluşturulabilir.

Bir grafın kendisini G, düğümlerini V ve kenarlarını E ifade etmek üzere, bir grafın düğümleri ve kenarları G=(V,E) ifadesi ile temsil edilir. Her bir düğüm "v" ve her bir kenar "e" olmak üzere, PR(v) negatif olmayan bir başlangıç pagerank değerine sahiptir. n, u'dan giden bağlantıların sayısı olmak üzere, her bir e = (u, v) kenarı 1/n ile ağırlıklandırılır.

$$PR(A)=\sum_{i=1}^{n}\frac{PR(i)}{C(i)} \tag{4.1}$$

Eş. 4.1'de yer alan $PR(A)$, A düğümünün PageRank değerini, $PR(i)$ i. düğümün pagerank değerini, $C(i)$ ise i. düğümden giden kenarların sayısını ifade eder. 1'den n'e kadar olan

düğümler, A'yı işaret eden bağlantıları içeren tüm graf düğümleridir (Ahmed ve Thomo, 2021). Pagerank değerlerinin hesaplanmasında bazı kaynaklarda (değeri genellikle 0,85 olarak atanan) bir sönümleme faktörü (damping factor) de bulunur. Eş. 4.2'de yer alan d ifadesi sönümleme faktörünü temsil etmektedir (Needham ve Hodler, 2019: 100).

$$PR(A)=(1\text{-}d)+d\left(\sum_{i=1}^{n}\frac{PR(i)}{C(i)}\right) \tag{4.2}$$

i. düğümü ziyaret eden herhangi bir gezinmenin, i. düğümün bağlantılı olduğu diğer düğümleri ziyaret etme olasılığı $1/C(i)$ olup M matrisi ile temsil edilebilir. Diğer taraftan $PR(i)$ değerleri V vektörü ile ifade edilebilir. Bu durumda (notasyon sadeliği amacıyla sönümleme faktörünün ihmal edildiği) Eş. 4.1, M.V olarak matris formunda yazılabilir. "t" iterasyon sayısı olmak üzere, M.V eşitliği aşağıdaki şekilde de ifade edilebilir.

$$V_{t+1}=M.V_t \tag{4.3}$$

Eş. 4.3'te yer alan $V_{t+1}$ vektörü, her iterasyonda tüm düğümler için tekrar hesaplanan pagerank değerlerini tutar. Ardışık iterasyonlar sonucunda $V_{t+1}$ ile $V_t$ vektörü arasındaki fark çok küçük olduğunda, algoritmanın nihai pagerank değerlerine yakınsadığı kabul edilir ve algoritma durdurulur.

Eş. 4.1'den (ya da 4.2'den) de açıkça anlaşılacağı üzere, bir A düğümünün pagerank değeri, ona işaret eden düğümlerin pagerank değerine bağlıdır. A düğümüne işaret eden düğümlerin pagerank değeri de yine A düğümlerinin pagerank değerlerine dayanır. Dolayısıyla düğümlerin pagerank değerlerine ilişkin başlangıç sorununun üstesinden gelmek için her düğüme tahmini bir pagerank değeri atanır. Bu vektör ise $V_0$ ile temsil edilebilir (Ahmed ve Thomo, 2021). Diğer bir ifadeyle t=1 iterasyonunda, Eş. 4.3'te yer alan $V_t$ vektörü $V_0$ vektörüne eşit olur. $V_t$ vektörü, takip eden iterasyonlarda güncellenmek suretiyle nihai pagerank vektörü elde edilir.

### 4.2.2. Arasındalık

Arasındalık (betweenness) merkeziliği, graftaki tüm düğüm çiftleri arasında en kısa yola sahip düğümün kullanılma sayısının hesaplanmasıdır. Arasındalık merkeziliği, farklı düğüm gruplarını birbirine bağlayan kritik noktaları bulmada etkilidir. Algoritmanın döndürdüğü

değer ne kadar büyükse düğüm de o kadar önemlidir. Sosyal bir ağda, farklı sosyal gruplarla en çok bağlantı kuran kişinin tespit edilmesi örnek verilebilir (Bechberger ve Perryman, 2020: 291).

Arasındalık merkeziliği, her düğüm için bir değer hesaplar ve böylece graftaki düğümleri sıralamaya olanak tanır. Bir v düğümü için arasındalık merkeziliği Eş. 4.4'te gösterildiği gibi hesaplanır.

$$\mathrm{BC(v)}=\sum_{s\neq v\neq t\in V}\frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4.4}$$

$\mathrm{BC(v)}$ ifadesi v düğümünün arasındalık değerini, $\sigma_{st}$ ifadesi s ve t düğümleri arasındaki en kısa patika sayısını, $\sigma_{st}(v)$ ifadesi v düğümünden geçen en kısa patikaların sayısını temsil eder. Eş. 4.4'ten bir düğümün arasındalık değerinin, düğümden geçen en kısa patikaların sayısıyla doğru orantılı olduğu anlaşılmaktadır. Diğer bir ifadeyle s ve t düğümleri arasındaki en kısa patika sayısının v düğümüne kıyasla artması, v düğümünün arasındalık değerini (önemliliğini) düşürecektir.

Büyük ölçekli gerçek dünya grafları, düzensiz bellek erişim modelleri ve düğümlerin çarpık dağılımı nedeniyle iş yükü dengesizliği gibi birçok zorluğu beraberinde getirir. Diğer taraftan çoğu yararlı arasındalık algoritmasının oldukça yüksek bir hesaplama karmaşıklığı / maliyeti vardır. Ağırlıklandırılmamış bir grafta arasındalık algoritması, V düğüm sayısı olmak üzere, $\mathrm{O(V^3)}$ karmaşıklığına sahiptir. Brandes'in önerdiği arasındalık tekniği ise E kenar sayısı olmak üzere, $\mathrm{O(V.E)}$ çalışma zamanı maliyetine sahiptir. Ayrıca grafa yeni düğümler ve / veya kenarlar eklendiğinde ya da silindiğinde, grafın yapısı değişeceği için arasındalık değerinin de sürekli olarak tekrar hesaplanması gerekecektir (Chernoskutov, Ineichen ve Bekas, 2015). Bu sebeplerle büyük ölçekli verilerde arasındalık merkeziliği yüksek hesaplama maliyetlerine sahip olduğundan, arasındalığın daha gelişmiş yakınsak türlerine başvurulabilir.

### 4.2.3. Derecelendirme

Bir düğüme gelen kenarların sayısını temsil eden derecelendirme (degree) algoritması anlaşılması en basit algoritmadır. Bu algoritma, düğümleri kenar sayılarına göre sıralar. Derece merkeziliği, iç derece ve dış derece ayrı ayrı ölçülerek çeşitlendirilebilir. Özellikle

ortalama, minimum ve maksimum değerler hesaplanırken bir grafın bağlantılılık seviyesini belirlemek için genellikle bir temel (baseline) olarak kullanılır. Sosyal ağlarda, en çok arkadaşa sahip düğümün hesaplanması örnek verilebilir. (Bechberger ve Perryman, 2020: 290-291).

### 4.2.4. Yakınlık

Yakınlık (closeness) merkeziliği, bir düğümden diğer tüm düğümlere giden en kısa yolun ortalama uzunluğunu ifade eder. Bu yüzden algoritmanın döndürdüğü değer ne kadar küçükse düğüm o kadar önemlidir. Bir düğümün diğer düğümlere göre merkezilik seviyesini gösterir. Sosyal bir ağda, kimlerin sosyal ağın "kalbinde" olduğunu tanımlar (Bechberger ve Perryman, 2020: 291). Yakınlık algoritması aşağıda Eş. 4.5'te gösterildiği şekilde hesaplanır.

$$C(v)=\sum_{u=1}^{n-1} d(v,u) \tag{4.5}$$

Eş. 4.5'te yer alan v ifadesi düğümü, n ifadesi grafta bulunan düğüm sayısını, d(v,u) ise u ve v düğümleri arasındaki en yakın patikayı temsil eder. $C(v)$ değerini, en kısa patikaların toplamından ziyade ortalama uzunluğunu temsil edecek şekilde normalleştirmek daha yaygındır. Böylece Eş. 4.6 kullanılarak farklı boyutlardaki graf düğümlerinin yakınlık merkeziliği karşılaştırılabilir (Needham ve Hodler, 2019: 84-85).

$$C_{norm}(v)=\frac{\sum_{u=1}^{n-1} d(v,u)}{n-1} \tag{4.6}$$

### 4.2.5. Özdeğer vektörü

Özdeğer vektörü (eigenvector) merkeziliği, belirli bir düğümün önemini hesaplamak için girdi olarak komşu düğümlerin göreli önemini kullanır. Bir düğümün çok sayıda diğer düğümlere bağlı olması, onun önemli olduğu anlamına gelmez. Bunun yerine, düğümün komşularının önemi, düğümün genel önemini hesaplamak için kullanılır. Bir düğümün komşu düğümleri kendi aralarında nispeten bağlantısızsa, yüksek seviyede bağlantılı olan

daha az komşu düğüme sahip bir düğümden daha düşük bir değer alır (Bechberger ve Perryman, 2020: 291).

### 4.3. Topluluk Algılama Algoritmaları

Kümeleme (clustering) ya da bölümleme (partitioning) algoritmaları olarak da adlandırılan topluluk algılama (community detection) algoritmaları, üyelerin grup (küme, topluluk, bölüm) içindeki düğümlerle ilişkilerinin, grup dışındaki düğümlerle ilişkilerinden daha önemli olduğu fikrine odaklanır. Bu ilişkilerin belirlenmesi, düğüm kümelerini, yalıtılmış grupları ve ağ yapısını ortaya çıkarır. Böylece akran gruplarının benzer davranışlarını ya da tercihlerini anlamaya, esnekliği (resiliency) tahmin etmeye, gömülü ilişkileri bulmaya ve diğer analizler için veri hazırlamaya yardımcı olur. Topluluk algılama algoritmaları, veri analizi için ağ görselleştirmesi amacıyla da yaygın olarak kullanılır.

Topluluk algılama algoritmalarında ilişkilerin yoğunluğu çok önemlidir. Çok yoğun bir grafta düğümler sınırlı sayıda küme oluşturabilir. Bu sorunu çözmek amacıyla dereceye, ilişki ağırlıklarına veya benzerlik ölçütlerine göre filtreleme yapılarak küme sayısı arttırılabilir. Diğer taraftan çok seyrek bir grafta neredeyse her düğüm ayrı bir küme oluşturabilir. Bu durumda ise daha alakalı bilgiler taşıyan ek ilişki türleri dahil edilebilir (Needham ve Hodler, 2019: 109-11).

Bu algoritmalar beş temel başlık altında incelenebilir.

- Üçgen Sayısı (Triangle Count) ve Kümeleme Katsayısı (Clustering Coefficient)
- Güçlü Bağlanan Bileşenler (Strongly Connected Components)
- Zayıf Bağlanan Bileşenler (Weakly Connected Components)
- Etiket Yayılımı (Label Propagation)
- Louvain Modülerliği (Louvain Modularity)

#### 4.3.1. Üçgen sayısı ve kümeleme katsayısı

Üçgen sayısı (triangle count) ve kümeleme katsayısı (clustering coefficient) algoritmaları genellikle birlikte kullanılırlar.

Üçgen sayısı, graftaki her bir düğümden geçen üçgenlerin sayısını ifade eder. Üçgen, birbiriyle bağlantılı üç düğümden oluşan bir kümedir. Üçgen sayısı, genel veri kümesini değerlendirmek için global olarak da çalıştırılabilir.

Kümeleme katsayısı algoritmasının amacı, bir grubun kümelenme yoğunluğunu ölçmektir. Algoritmanın hesaplanmasında üçgen sayısı kullanır. Maksimum puanı ifade eden 1 değeri, belirli bir grup içindeki tüm düğümlerin birbiriyle bağlantılı olduğunu belirtir.

Bir grubun kararlılığını belirlemek ya da kümeleme katsayısı gibi diğer ağ ölçümlerini hesaplamak için üçgen sayısından yararlanılır. Üçgen sayısı, sosyal ağ analizinde yaygın bir şekilde kullanılır. Kümeleme katsayısı, rassal olarak seçilen düğümlerin bağlanma olasılığını verir. Belirli bir grubun veya genel olarak ağın üyelerinin birbirine bağlılığını hızlı bir şekilde değerlendirmek amacıyla kullanılabilir. Ayrıca üçgen sayısı ve kümeleme katsayısı algoritmaları birlikte, esnekliği (resiliency) tahmin etmek ve ağ yapılarını aramak için kullanılır (Needham ve Hodler, 2019: 109-111, 116).

Kümeleme katsayısı iki kategoride incelenebilir: yerel kümeleme ve global kümeleme.

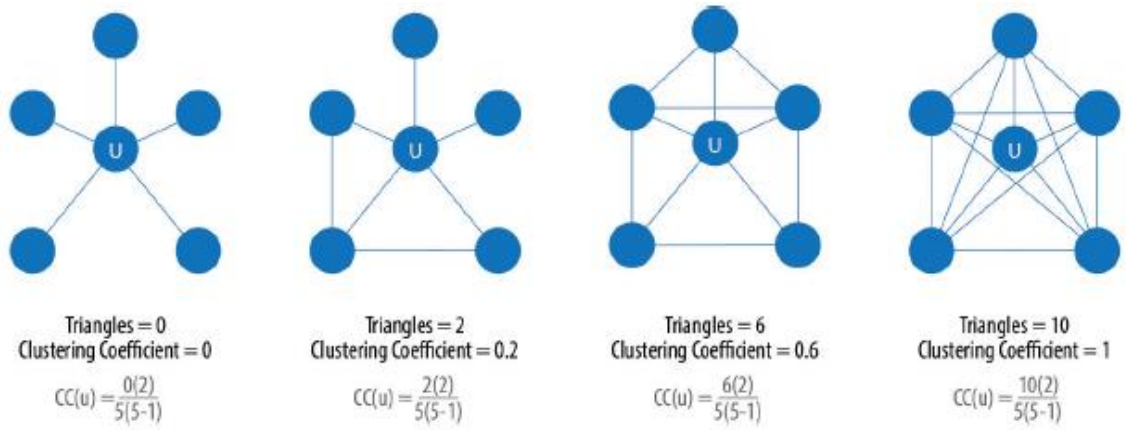
Yerel kümeleme katsayısı

Yerel kümeleme katsayısı, bir düğümün komşularıyla bağlantılı olma olasılığıdır. Bu puanın hesaplanması üçgen sayısı kullanılır. Yerel kümeleme (clustering coefficient) katsayısı aşağıdaki eşitlikte gösterildiği gibi hesaplanır.

$$\mathrm{CC(u)}=\frac{2\mathrm{R_u}}{\mathrm{k_u(k_u\text{-}1)}} \tag{4.7}$$

Eş. 4.7'de yer alan u ifadesi düğümü, $\mathrm{R_u}$ ifadesi u'nun komşu düğümlerinden geçen ilişkilerin sayısı (yani u'dan geçen üçgenlerin sayısı), $\mathrm{k_u}$ ise u'nun derecesini temsil eder. $\mathrm{k_u}$-1 ifadesi de gruptaki maksimum ilişki sayısını belirtir (Needham ve Hodler, 2019: 116).

Beş ilişkiye sahip bir düğüm için farklı üçgenler ve kümeleme katsayıları Şekil 4.5'te gösterilmektedir.




Şekil 4.5. u düğümü için üçgen sayısı ve kümeleme katsayısı, (Needham ve Hodler, 2019: 115)

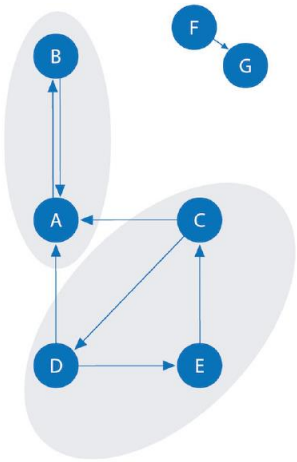
Şekil 4.5'te beş ilişkiye sahip bir düğüm kullanıldığı için kümeleme katsayısı, her zaman üçgen sayısının %10'una eşit olur. İlişki sayısı değiştirildiğinde, örneğin 6 kabul edildiğinde, ikinci ve üçüncü örnekler için (üçgen sayısının 2 ve 6, yani aynı kabul edildiği varsayımında) kümeleme katsayıları sırasıyla 0.13 ve 0,40 olur.

Global kümeleme katsayısı

Global kümeleme katsayısı, yerel kümeleme katsayılarının normalize edilmiş toplamıdır. Kümeleme katsayıları, her düğümün diğer düğümlerle olan ilişkisini ortaya çıkarmak için etkili bir araçtır. Ayrıca ilişki seviyelerini ayarlamak için örneğin düğümlerin %40 bağlı olduğu yerler şeklinde eşikler de belirlenebilir (Needham ve Hodler, 2019: 116).

### 4.3.2. Güçlü bağlanan bileşenler

Güçlü bağlanan bileşenler (strongly connected components, SCC) algoritması, en eski graf algoritmalarından biridir. SCC yönlü (directed) bir grafta bağlantılı düğüm kümelerini bulur. Her bir düğüme aynı kümedeki diğer bir düğümden çift yönlü olarak erişilebilir. Çalışma zamanı işlemleri, düğüm sayısıyla orantılı olarak ölçeklenir.




Şekil 4.6. Güçlü bağlanan bileşenler,
(Needham ve Hodler, 2019: 119)

Şekil 4.6'da, bir SCC grubundaki düğümlerin doğrudan komşu olması gerekmediği, ancak kümedeki tüm düğümler arasında yönlü kenarlar olması gerektiği görülebilir. Birbirine güçlü bağlanan iki bileşen gölgeli olarak gösterilmiştir: {A,B,} ve {C,D,E}. Alt tarafta kalan bileşenin elemanları olan C,D,E düğümleri, bazı durumlarda önce başka bir düğümden geçmeleri gerekse de üst tarafta yer alan diğer bileşenin düğümlerine ulaşabilir.

SCC algoritması, bir grafın yapılandırılma biçimini ya da kümelerin yoğunlaşma seviyesini yüzeysel olarak analiz etmek amacıyla kullanılabilir. Benzer şekilde öneri motorları gibi uygulamalar için bir gruptaki benzer davranış veya eğilimlerin profilini çıkarmada da kullanılabilir (Needham ve Hodler, 2019: 119-120).

SCC, dahili bellek kullanan graflarda, örneğin tek bilgisayardan çalıştırılan bir graf veri tabanında, DFS ile birlikte verimli bir şekilde kullanılabilir. Ancak dağıtık veya harici bellek kullanan graflarda SCC ile DFS'yi kombinlemek (DFS'den dolayı) çok verimsiz olacaktır (Kyrola, 2014: 15).

### 4.3.3. Zayıf bağlanan bileşenler

Bağlı bileşenler (connected components) ya da birlik bulma (union find) olarak da adlanırılan zayıf bağlanan bileşenler (weakly connected components, WCC) algoritması, yönsüz (undirected) bir grafta bağlantılı düğüm kümelerini bulur. Her bir düğüme aynı kümedeki diğer bir düğümden tek yönlü olarak erişilebilir. SCC algoritmasından farklıdır.

Çünkü WCC düğüm çiftleri arasında tek yönlü bir yola ihtiyaç duyarken, SCC iki yönlü bir yola ihtiyaç duyar.

SCC'de olduğu gibi WCC de genellikle bir grafın yapısını ana hatlarıyla analiz etmek amacıyla kullanılır. Etkin bir şekilde ölçeklendiğinden, sık güncelleme gerektiren graflarda kullanışlıdır. Gruplar arasında ortak yeni düğümleri hızlı bir şekilde gösterebildiğinden, sahtecilik tespiti gibi analizler için yararlıdır (Needham ve Hodler, 2019: 124).

Bir graftaki düğüm sayısı V olmak üzere, O(V) karmaşıklık zamanına sahip WCC, bir bilgisayarda depolanabilecekleri varsayımında, kenarlardan yalnızca tek bir geçişi garanti eden iyi bir birlik bulma algoritması kullanılarak çok verimli çalışabilir. Büyük ölçekli graflarda, her yinelemede düğüm komşularının minimum etiketlerine ve kendi düğümüne göre yeni bir etiket seçen yinelemeli bir minimum etiket yayılım (label propagation) algoritması kullanabilir (Kyrola, 2014: 15).

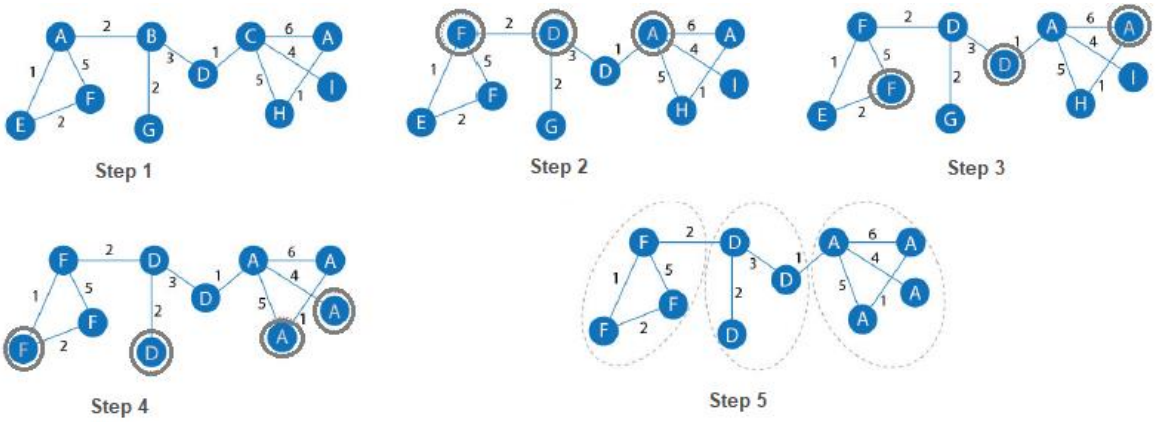
### 4.3.4. Etiket yayılımı

Etiket yayılımı algoritması (label propagation algorithm, LPA), bir grafta yer alan grupları hızlı bir şekilde bulabilir. LPA'da düğümler, doğrudan (yakın) komşularına göre gruplarını seçer. Bu algoritma tek bir etiketin, yoğun bir şekilde birbirine bağlı bir düğüm grubunda hızla baskın hale gelebileceği; seyrek bağlantılı bir bölgeyi geçmekte ise sorun yaşayacağı fikrine dayanır. Etiketler yayıldıkça, yoğun şekilde birbirine bağlı her bir düğüm grubu, tekil bir etiket üzerinde hızla bir fikir birliğine varır. Yayılımın sonunda sadece grupları temsil eden birkaç etiket kalacak ve aynı etikete sahip düğümler aynı gruba ait olacaktır. Algoritma, düğümlerin potansiyel olarak birden çok kümenin parçası olduğu çakışmaları, en yüksek birleşik kenar ve düğüm ağırlığını dikkate alarak çözer.

Bazı düğümlere çekirdek (seed) etiketler verildiğinde, etiketlenmemiş diğer düğümlerin çekirdek etiketleri benimseme olasılığı daha yüksek olacağından çözüm aralığı daralır. Bu metot toplulukları bulmak için yarı denetimli bir öğrenme yöntemi olarak kabul edilebilir. Yarı denetimli öğrenme, çok sayıda etiketlenmemiş verinin, az miktarda etiketlenmiş veri setiyle makine öğrenimi yöntemi kullanılarak sınıflandırılmasıdır. Topluluk sonuçları, belirgin bir şekilde benzer birkaç topluluk arasında sürekli gidip gelir ve algoritma hiçbir

zaman tamamlanmaz. Böyle bir durumda çekirdek etiketleri, birkaç topluluk arasında etiketi sürekli değişen düğümleri bir çözüme doğru yönlendirmeye yardımcı olur.

Etiket yayılımı algoritması çekme (pull) ve ve itme (push) şeklinde iki temel yönteme sahiptir. Ancak çekme yöntemi, paralelleştirmeye çok uygun olduğundan büyük ölçekli graflarda daha başarılıdır. Şekil 4.7'de, kenar ağırlıklarına dayanan çekme (pull) yöntemi gösterilmektedir.




Şekil 4.7. Etiket yayılımı: çekme yönteminin işleyişi, (Needham ve Hodler, 2019: 128)

Birinci adımda, kenarlarda yer alan değerler kenar ağırlıklarını temsil etmek üzere, opsiyonel olarak A çekirdek (seed) etiketi kullanılan iki düğüm dışında tüm düğümler tekil etiketlere sahiptir.

İkinci adımda, düğümlerin etiketleri en yüksek kenar ağırlıklarına göre değiştirilir. Örneğin birinci adımda A ile F arasındaki kenar ağırlığı 5, (A düğümünün diğer komşu kenarlarını ifade eden) A ile E ve A ile B arasındaki (sırasıyla 1 ve 2 olan) kenar ağırlarından daha büyük olduğu için A düğümün etiketi F olarak değiştirilmiştir. Benzer süreç birinci adımda etiketi B ve C olan diğer iki düğümde de işleyerek, B ve C düğümlerinin etiketleri D ve A etiketleriyle değiştirilmiştir.

Üçüncü adımda, en yüksek kenar ağırlıkları aynı etikete sahip olduğundan düğümlerin etiketlerinde herhangi bir değişiklik olmamıştır.

Dördüncü adımda algoritma, ikinci ve üçüncü adımdaki işlemleri tekrar ederek tüm düğümlerin etiketlerini değiştirir.

Beşinci adımda ise tüm düğümler, komşularının baskın etiketine sahip olduğu için algoritma sonlanmış ve üç küme belirlenmiştir.

Diğer algoritmaların aksine etiket yayılımı, aynı graf üzerinde birden çok kez çalıştırıldığında farklı topluluk yapıları döndürebilir. LPA'nın düğümleri değerlendirme sırası, geri döndürdüğü nihai topluluklar üzerinde etkili olabilir.

LPA paralelleştirilebilir olduğundan son derece hızlı bir yöntemdir. Bu yüzden büyük ölçekli graflarda, özellikle ağırlıklar bilindiğinde LPA kullanımı önerilir (Needham ve Hodler, 2019: 127-129).

### 4.3.5. Modülerlik

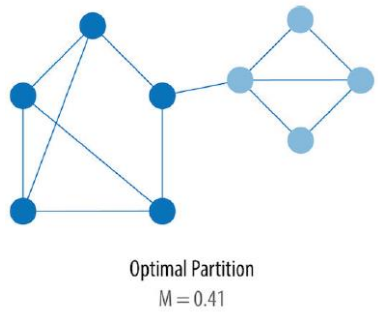
Modülerlik, bir küme içindeki kenarların yoğunluğuna bakılarak bir düğümün, ortalama veya rassal bir örneğe kıyasla, atandığı gruba uyumunun ölçüsü olarak ifade edilebilir. Diğer bir ifadeyle modülerlik, bir grafı modüllere (veya kümelere) böldükten sonra gruplamaların gücünü ölçerek toplulukları ortaya çıkarmaya yönelik bir tekniktir. Sadece bir küme içindeki kenarların yoğunluğuna bakmanın aksine, modülerlik, belirli kümelerdeki kenar yoğunluklarını, kümeler arasındaki kenar yoğunluklarıyla karşılaştırır. Bu gruplamaların kalitesinin ölçüsüne ise modülerlik denir.

Modülerlik algoritmaları, farklı gruplamaları test etmek ve grup büyüklüğünü artırmak için çoklu yinelemeler kullanır. Böylece toplulukları önce yerel, sonra da global olarak optimize eder. Bu strateji topluluk hiyerarşilerini belirler ve genel yapının kapsamlı bir şekilde anlaşılmasını sağlar (Needham ve Hodler, 2019: 134).

Çok sayıda modülerlik hesaplama yöntemi vardır. Basit bir modülerlik hesaplaması, verilen gruplar içindeki kenarların sıklığından, kenarların tüm düğümler arasında rassal dağılmış olması durumunda beklenen sıklığından çıkarılmasına dayanır. Değer her zaman 1 ile -1 arasındadır. Pozitif değerler beklenenden daha fazla ilişki yoğunluğunu, negatif değerler ise daha az yoğunluğu gösterir. Modülerliğin hesaplanmasında kullanılan temel parametrelerin anlaşılması amacıyla, Eş. 4.8’de sunulan basit bir modülerlik hesaplaması yönteminden yararlanılacaktır.

$$M=\sum_{c=1}^{n_c}\left[\frac{L_c}{L}-\left(\frac{k_c}{2L}\right)^2\right] \tag{4.8}$$

Eş. 4.8'de yer alan $n_c$ ifadesi topluluk / grup (community) sayısını, L tüm grupta yer alan kenar sayısını, $L_c$ bir grupta yer alan kenar sayısını, $k_c$ bir grupta yer alan düğümlerin toplam derecesini ifade eder. Bir grupta yer alan düğümlerin toplam derecesi, grupta yer alan her bir düğümün sahip olduğu kenar sayısının toplamına işaret eder.




Şekil 4.8. Modülerliğin hesaplanması,
(Needham ve Hodler, 2019: 135)

Eş 4.8'de sunulan modülerlik formülü kullanılarak Şekil 4.8'te yer alan temsili grafın modülerliği aşağıda gösterildiği gibi hesaplanır. Şekil 4.8'te koyu ve açık mavi ile temsil edilen grafın tümünde yer alan kenar sayısı L=13, grup sayısı ise (koyu ve açık mavi şeklinde iki grup olduğu için) $n_c$=2'dir. Koyu mavi olarak (solda) belirtilen grafın kenar sayısı $L_c$=7 ve düğümlerin toplam derecesi ise $k_c$=15'tir. Açık mavi olarak (sağda) belirtilen grafın kenar sayısı ise $L_c$=5 ve düğümlerin toplam derecesi ise $k_c$=11'dir.

Koyu mavi kısım: $$M_1=\frac{7}{13}-\left(\frac{15}{2(13)}\right)^2=0{,}205$$

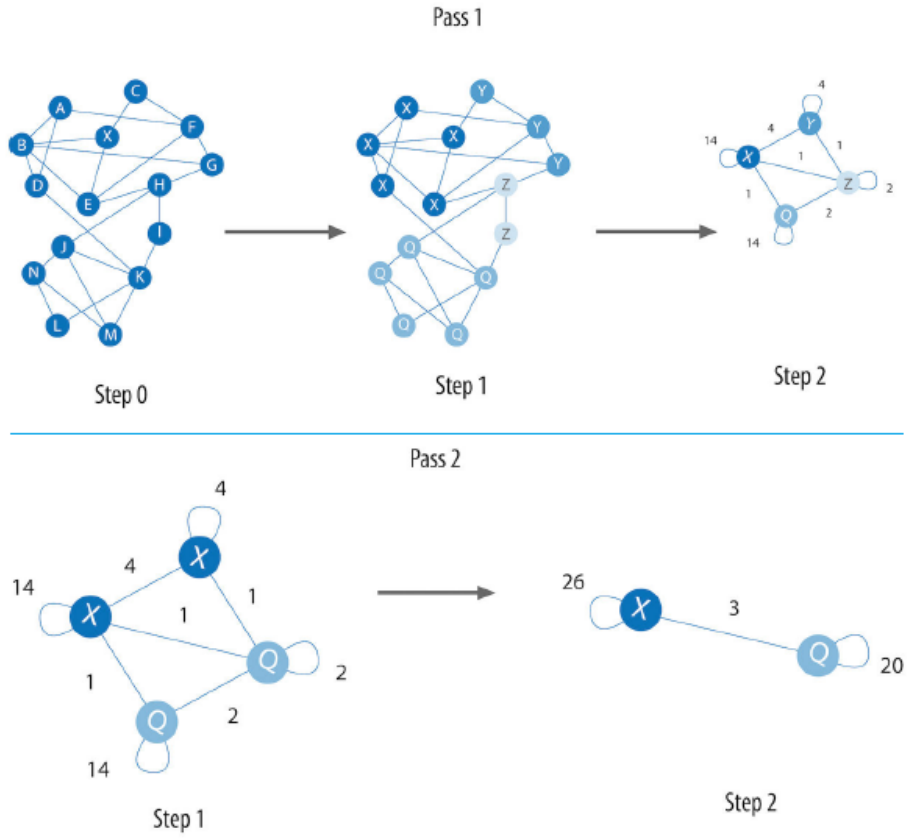
Açık mavi kısım: $$M_2=\frac{5}{13}-\left(\frac{11}{2(13)}\right)^2=0{,}206$$

Toplam: $$M=M_1+M_2=0{,}205+0{,}206=0{,}41$$

Louvain modülerliği

Louvain modülerlik (Louvain modularity) algoritması, düğümleri farklı gruplara atar ve onları topluluk yoğunluğuyla karşılaştırarak kümeler. 2008'de önerilen Louvain algoritması, modülerliğe dayalı en hızlı algoritmalardan biridir. Toplulukları tespit etmenin yanı sıra, farklı ölçeklerdeki topluluk hiyerarşilerini de ortaya çıkarır. Böylece bir grafın yapısı, farklı ayrıntı seviyelerinde analiz edilebilir.

Louvain modülerlik algoritması, başlangıçta küçük toplulukları oluşturmak için tüm düğümlerde modülerliği yerel olarak optimize eder. Sonrasında ise her bir küçük topluluk daha büyük kümeler halinde gruplandırılır ve global bir optimuma ulaşana kadar ilk adım tekrarlanır. Louvain algoritması, Şekil 4.9'da gösterildiği gibi her biri kendi içinde adımlara sahip iki sürecin tekrarlanan uygulamasından oluşur.




Şekil 4.9. Louvain algoritmasının işleyişi, (Needham ve Hodler, 2019: 136)

Şekil 4.9'ın üst kısmına karşılık gelen algoritmanın ilk aşamasında (pass 1), yerel modülerlik optimizasyonları için düğümler topluluklara atanır. İlk adımda (step 0) başlangıç düğümünün X düğümü olduğu varsayılmıştır. İkinci adımda (step 1) X düğümü, B, C ve E

düğümlerinden en yüksek modülarite değişimine yol açan düğüme katılır. Süreç her bir düğümün en yüksek modülarite değişimine yol açan düğümlere katılması şeklinde tekrar eder. Üçüncü adımda (step 2), önceki adımda 4 gruba ayrılmış düğümler X, Y, Z ve Q şeklinde toplulaştırılarak kendi süper düğümlerini (super node) oluşturur. Süper düğümleri birbirine bağlayan (ve süper düğümlerde bulunan düğüm sayılarına karşılık gelen) kenarlarda yer alan değerler kenar ağırlıkları olarak ifade edilir. Şekil 4.9'un alt kısmına karşılık gelen algoritmanın ikinci aşamasında (pass 2), bir önceki aşamanın (pass 1) çıktısı süper düğümler esas alınarak, daha büyük süper düğümler oluşturulmak amacıyla ilk aşamadaki (pass 1) adımlar (step 0, step 1 ve step 2) tekrarlanır. Bu iki aşama, toplulukların modülerliğini artıran yeniden atamaların yapılamamasına kadar tekrarlanır.

Optimizasyonun ilk aşamasında (pass 1) Louvain algoritması, bir grubun modülerliğini hesaplamak için aşağıda yer alan formülü kullanır.

$$Q=\frac{1}{2m}\sum_{u,v}\left[A_{uv}-\frac{k_u k_v}{2m}\right]\delta(c_u,c_v) \tag{4.9}$$

Eş. 4.9'da yer alan u ve v birer düğümü, m grafın tamamındaki kenarların toplam ağırlığını, dolayısıyla 2m ifadesi ise modülerlik formüllerinde kullanılan ortak bir normalizasyon değerini ifade eder. $\left[A_{uv}-\frac{k_u k_v}{2m}\right]$ ifadesi u ve v düğümleri arasındaki ilişkinin gücünü temsil eder. $A_{uv}$ ifadesi u ve v arasındaki kenarın ağırlığını, $k_u$ ifadesi u'nun kenar ağırlıkları toplamını, $k_v$ ise v'nin kenar ağırlıkları toplamını temsil eder. $\delta(c_u,c_v)$ ifadesi ise u ve v anı kümede ise 1, farklı kümede ise 0 değerini alır.

Louvain modülerliği, büyük ölçekli graflarda toplulukları bulmak için tercih edilebilir. Çünkü Louvain algoritması, karmaşıklık zamanı yüksek olan tam modülerliğin aksine, keşifsel (heuristic) bir yöntem uyguladığı için standart modülerlik algoritmalarının zorlanabileceği büyük ölçekli graflarda başarılıdır.

Ayrıca Louvain algoritması karmaşık ağların yapısını değerlendirmek, özellikle de pek çok hiyerarşi düzeyini ortaya çıkarmak için çok faydalıdır. Algoritma, farklı ayrıntı düzeylerini mercek altına almaya olanak tanır ve alt toplulukların alt toplulukların alt topluluklarını tespit eden derinlemesine sonuçlar sağlayabilir (Needham ve Hodler, 2019: 136).

# 5. SONUÇ

Büyük veri günümüzün potansiyel açıdan katma değeri en yüksek çalışma alanlarından biri olarak öne çıkmaktadır. Ancak büyük verinin hacim, çeşitlilik, hız gibi karakteristikleri sebebiyle, geleneksel veri tabanları kullanılarak ondan değer yaratma süreci çeşitli zorluklar içermektedir. Büyük verinin hacmi göz önüne alındığında, graf veri tabanları, büyük ölçekli verilerde CRUD operasyonlarını etkin bir şekilde gerçekleştirmeye olanak tanır. Bu yüzden graflar, büyük veriden değer yaratmanın bir adresi olarak gösterilmektedir. Diğer taraftan NoSQL veri tabanlarından olan graf veri tabanları, nesneler arasındaki ilişkilere odaklandığından gerçek dünya verilerini modellemede diğer veri tabanlarına kıyasla daha başarılıdır. Başlangıçta sosyal medya ağlarının modellenmesi amacıyla kullanılan graflar, son yıllarda, nesnelerin interneti (internet of thing, IoT), bilgi (knowledge) grafları, ulaşım ağları, semantik web, bağlantılı açık veri (linked open data, LOD), iletişim, lojistik, veri merkezi yönetimi, biyoinformatik gibi özellikle de karmaşık tekrarlı sorguların ve büyük ölçekli verinin söz konusu olduğu hemen her alanda yaygın bir şekilde kullanılmaktadır. ACID özelliklerini sağlayan Neo4j gibi bazı graf veri tabanlarının, ilişkisel veri tabanlarının etki alanına doğru genişlemesi dikkat çekmektedir.

Graf veri tabanları, büyük ölçekli verilerin işlenmesine ve karmaşık tekrarlı sorguların makul bir şekilde gerçekleştirilmesine ilişkin son 15 yılda ortaya çıkan ihtiyaçları karşılamada şaşırtıcı derecede başarılıdır. Verilerdeki bağlantıları zengin ilişki temsillerini kullanarak ortaya çıkartan graflar, özellikle de ilişkisel veri tabanlarının baş edemediği büyük ölçekli verilerin işlenmesinde tercih edilir. Bununla birlikte graf veri tabanları sadece büyük ölçekli verilerde değil, ilişkisel veri tabanlarında birleştirme (join) işlemlerinden dolayı performansı önemli ölçüde kısıtlayan, yüksek dereceli sorgularla baş etmek amacıyla da kullanılmaktadır.

İndekssiz komşuluk kuralına dayanan doğal graf işleme teknolojisini kullanan graflar, sorguları grafın tamamı yerine belirli bir parçasında (alt graflarda) gerçekleştirdiğinden, karmaşıklık zamanı tüm grafın büyüklüğüyle değil, sorgunun gerçekleştirilmesi amacıyla gezinilen alt grafın büyüklüğüyle orantılı değişir. Bunun bir sonucu olarak, veri tabanının ölçeği büyüse de diğer veri tabanlarına kıyasla, graflarda veri işleme performansı sabit kalma eğilimdedir.

Doğal graf depolama teknolojisi ise her bir işlemi eş anlı (paralel) bir şekilde yürütebildiklerinden, graf algoritmalarının hızlı geçişlerini destekleyerek büyük ölçekli graflarda performansı önemli ölçüde arttırır. Ancak graf veri tabanlarında grafın boyutu ile performans ve depolama maliyetleri arasında bir değiş-tokuş (trade off) olduğundan, yüksek performans ve düşük depolama maliyetleri için graf boyutu (depolama kapasitesi) sınırlandırılmak suretiyle optimize edilebilir.

Gelişmiş graf işleme yöntemlerine esas teşkil eden yaklaşımlar olarak değerlendirilebilen temel graf işleme teknikleri, Apache Tinkerpop projesi kapsamında geliştirilen Gremlin graf sorgulama dili kullanılarak, uygulama düzeyinde de incelenmiştir. Bu kapsamda grafların en önemli yapılarından biri olan patikalar, grafların en güçlü özelliklerinden olan tekrarlı gezinmeleri (recursive traversal) kullanır. Patikalar, başlangıç düğümünden bitiş düğümüne götüren alternatif yolların (bağlantılı düğümlerin ve kenarların) bulunmasını sağlar. Ancak gezinilen yollar üzerindeki düğümler ve kenarlara ilişkin adım (gezinme) sayısını bilmezler. Bilinen yürüyüş (known walk) tekniğinde ise gezilecek kenar ve düğümler ile birlikte adım (gezinme) sayısı da bilinir. Böylece bilinen yürüyüş, patikaların optimize edilmesini, diğer bir ifadeyle başlangıç ve bitiş düğümleri arasındaki en kısa yolların tespit edilmesi suretiyle sorguların hızlı bir şekilde gerçekleştirilmesini sağlar. Diğer bir ifadeyle bilinen yürüyüş, CRUD operasyonlarının temeli olan gezinmelerin maliyetini düşürmek suretiyle veri işleme performansını arttırır.

Düğümlerin ve / veya kenarların benzerliklerine göre kategorize edilmesini ifade eden grup etiketleri (generic labels), graflarda gezinme performansını arttıran önemli yapılardan bir diğeridir. Ancak gereğinden fazla grup etiketi kullanmanın gezinme hızını düşüreceği daima göz önünde bulundurulmalıdır. Bu yüzden grup etiketi kullanmak bir optimizasyon yaklaşımı gerektirir.

Okuma performansını arttırmak için yazma anında verinin (graf üzerinde) farklı konumlara kopyalanması şeklinde tanımlanan denormalizasyon (denormalization), okuma ve yazma operasyonları arasında bir değiş-tokuş (tradeoff) içerir. Bu yönüyle grup etiketleri gibi denormalizasyon tekniği de bir optimizasyon yaklaşımı gerektirir. Okuma yoğun işlemler ya da okuma performansı zayıf graflar, bilginin veri tabanından çağrılması aşamasında ilave ağ, hafıza ve disk erişimi gerektirdiğinden, özellikle de dağıtık sistemlerde verimsiz

süreçlere yol açar. Söz konusu verimsiz süreçlerin çözümü için denormalizasyon tekniği önemli bir fark yaratacaktır.

Alt graflar (subgraphs), graf işleme performansı üzerinde en önemli etkiye sahip teknikler arasındadır. Düğümleri ve kenarları global ya da daha büyük bir graftan konuya özgü bir şekilde türetilen alt graflar, sorgunun konusu dışındaki düğümler ve kenarlar üzerinde gezinmeden kaçınmaya olanak tanıdığından, gezinmelerin maliyetini ciddi ölçüde düşürür. Alt graflar, bir graf veri tabanının etkin bir şekilde ölçeklenmesi amacıyla sıklıkla kullanılır. Bu durum ise grafların diğer veri tabanlarından farklı olarak ölçeklendiğine işaret eder. Ancak yazma aşamasında ilgili alt grafların tümünün güncellenmesi gerektiği için çok fazla alt grafla çalışmak, graf veri tabanı üzerinde önemli bir yük oluşturabilir. Bu yüzden grup etiketleri ve denormalizasyon tekniği gibi alt grafların da bir optimizasyon yaklaşımı gerektirdiği akılda tutulmalıdır.

Kaçış odası metaforunda incelendiği üzere, mümkün olduğunca herhangi bir düğümde kalmak ya da gezinmenin düğümler yerine kenarlar üzerinde gerçekleştirilmesi, onların gezinme hızını, dolayısıyla CRUD operasyonlarının performansını önemli ölçüde arttırmaktadır. Uygulama aşamasında graf işleme teknikleri kullanılarak yapılacak performans optimizasyonunun, graf veri tabanının temel işlevselliği tesis edildikten sonra gerçekleştirilmesi önerilir. Temel işlevselliğin tesis edilmesi ise veri tabanının üretim benzeri kapsamlı bir test veri setiyle iş kurallarına göre çalıştırılmasını ifade eder.

Gelişmiş graf işleme yöntemleri çerçevesinde, grafların en temel operasyonlarını patika bulma ve arama algoritmaları gerçekleştirir. Patika bulma ve arama algoritmalarından genişlik öncelikli arama (BFS) algoritması, öncelikli olarak, bir düğümün birinci dereceden komşularını keşfetmeye odaklandığından, graf üzerinde yatay yönde gezinir. Temel bir algoritma olan BFS'nin doğrudan kullanımına sık rastlanmasa da hedefe yönelik diğer algoritmaların tasarımında ya da onlarla birlikte yaygın şekilde kullanılır. Derinlik öncelikli arama (DFS) algoritması ise graf üzerinde öncelikli olarak dikey yönde gezinir. Grafın yapısını ve / veya sorgunun amacına göre daha performanslı bir gezinme için BFS ya da DFS algoritmasından uygun olanı tercih edilebilir. İki düğüm arasındaki en kısa yolu tespit etmek amacıyla, fazladan gezinmeler yapabilen Dijkstra algoritması yerine, ondan türetilen ve onun daha gelişmiş bir versiyonu olan A* algoritması ön plana alınabilir. Rassal yürüyüş

algoritması ise genellikle, rassal bir alt grafa ihtiyaç duyulduğunda, diğer algoritmalarla birlikte kullanılır.

Grafların en güçlü yanlarından bir diğeri merkezilik algoritmaları ise graf içindeki düğümlerin önemlilik derecelerinin saptanmasına olanak tanır. Google'ın arama motorunda da kullanılan pagerank algoritması, bir düğümün önemlilik derecesini diğer önemli düğümlerden kendisine gelen bağlantılar / kenarlar üzerinden ölçer. Olasılıksal süreçler içeren pagerank algoritması, rassal yürüyüş algoritmasıyla birlikte çalışır. Arasındalık merkeziliği, farklı düğüm grupları arasında köprü görevi gören düğümleri bulmada etkilidir. Düğümleri kenar sayılarına göre sıralayan derecelendirme algoritması ortalama, minimum ve maksimum değerler gibi toplulaştırma operasyonlarında sıklıkla kullanılır. Yakınlık (closeness) merkeziliği ise bir düğümden diğer tüm düğümlere giden en kısa yolun ortalama uzunluğunu üzerinden düğümün önemini ölçer. Özdeğer vektörü ise bir düğümün önemini hesaplamak için diğer düğümlerle olan bağlantı sayısına odaklanmak yerine, komşu düğümlerin önemini esas alır.

Farklı merkezilik algoritmaları, ölçülmek istenen etkiye bağlı olarak önemli ölçüde farklı sonuçlar üretebilir. Bu yüzden probleme en uygun merkezilik algoritmasının seçilmesi önemlidir. Merkezilik algoritmalarının birçoğu, küçük ve orta ölçekli graflar için her bir düğüm çifti arasındaki en kısa patikaları hesaplamada kullanılabilir. Ancak karmaşıklık zamanları, büyük ölçekli graflar için problem olabilir. Büyük ölçekli graflarda uzun çalışma sürelerinden kaçınmak için bazı algoritmaların (örneğin, arasındalık (betweenness) algoritması gibi) yakınsak versiyonları tercih edilebilir (Needham ve Hodler, 2019: 77-78).

Topluluk algılama (community detection) algoritmaları, üyelerin grup içindeki düğümlerle ilişkilerine odaklanır. Benzer özellikleri sahip düğümleri, diğer düğüm kümelerinden yalıtarak grafların analizini kolaylaştırır. Bunlardan kümeleme katsayısı, belirli bir grubun veya genel olarak ağın üyelerinin birbirine bağlılığını hızlı bir şekilde değerlendirmek için tercih edilebilir. Yönlü bir grafta bağlantılı düğüm kümelerini bulan güçlü bağlanan bileşenler (SCC), tek bilgisayardan çalıştırılan büyük ölçekli bir graf veri tabanında, DFS ile birlikte verimli bir şekilde kullanılabilir. Yönsüz bir grafta bağlantılı düğüm kümelerini bulan zayıf bağlanan bileşenler (WCC), SCC'nin aksine, düğümler arasında tek yönlü bir yol üzerinden gezinir. WCC, tek bilgisayardan çalıştırılan büyük ölçekli bir graf veri tabanında, yinelemeli bir minimum etiket yayılım algoritmasıyla birlikte kullanılabilir.

Çekirdek etiketlerle kullanıldığında, bir tür denetimsiz makine öğrenme algoritmasına dönüşen etiket yayılımı, paralelleştirmeye yatkın olduğundan son derece hızlı bir algoritmadır. Düğümleri / kenarları ağırlıklandırılmış graflarda kullanılabilen etiket yayılımı algoritması büyük ölçekli verilerin işlenmesinde sıklıkla tercih edilir. Graf içindeki grupları / kümeleri ortaya çıkarmayı amaçlayan modülerliğin karmaşıklık maliyeti yüksektir. Ancak keşifsel (heuristic) bir yaklaşımı benimseyen ve nispeten düşük bir karmaşıklık maliyetine sahip Louvain modülerliği, büyük ölçekli graflarda toplulukları tespit etmek amacıyla ön planda tutulabilir.

Topluluk algılama algoritmaları genellikle "grupları belirlemek" şeklinde ortak bir amaca sahiptir. Ancak farklı algoritmalar farklı varsayımlarla başladıkları için farklı toplulukları ortaya çıkarabilirler. Bu yüzden belirli bir problem için doğru algoritmayı seçmek için deneme-yanılma yöntemine başvurmak gerekir. Topluluk algılama algoritmalarının çoğu, çevrelerine kıyasla grup içi ilişki yoğunluğu yüksek olduğunda makul ölçüde iyi sonuçlar verir. Ancak gerçek dünyadaki ağlar genellikle daha az belirgindir. Bu çerçevede bulunan toplulukların doğruluğu, sonuçları bilinen topluluklarla kıyaslanarak sınanabilir (Needham ve Hodler, 2019: 143).

Büyük ölçekli graf verilerinin işlenmesine ilişkin zorluklardan biri, graf sorgularının verimli bir şekilde değerlendirilmesidir. Uygulamada, mevcut graf veri tabanları mimarisinin çoğu tipik olarak tek bir makinede (kümelenmemiş) çalışacak şekilde tasarlanmıştır. Bu nedenle graflar, graf verilerinin sınırlı bir alt kümesini temsil eden OLTP türü sorguları etkin bir şekilde işleyebilir. Ancak graf veri tabanları, grafın büyük bölümlerinde gezinmenin, karmaşık birleştirmelerin ve toplulaştırmaların gerekli olduğu karmaşık OLAP türü analiz işlemlerinde en iyi çözüm olmayabilir (Ragab, 2020).

# KAYNAKLAR


Ahmed, A. ve Thomo, A. (2020). PageRank for Billion-Scale Networks in RDBMS. *International Conference on Intelligent Networking and Collaborative Systems*, 1263(2020), 89-100. doi:10.1007/978-3-030-57796-4_9

Asiler, M., Yazıcı A., George, R. (2022). HyGraph: A Subgraph Isomorphism Algorithm for Afficiently Querying Big Graph Databases. *Journal of Big Data*, 9(40), 1-22. doi: 10.1186/s40537-022-00589-0

Bechberger, D. ve Perryman, J. (2020). *Graph Databases in Action*. Shelter Island, NY: Manning Publications Co.

Besta, M., Gerstenberger, R., Peter, E., Fischer, M., Podstawski, M., Barthels, C., Alonso, G. ve Hoefler, T. (2022). Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *arXiv*, 09017(6)., 1910-1951. https://doi.org/10.48550/arXiv.1910.09017

Chernoskutov, M., Ineichen, Y. ve Bekas, C. (2015). Heuristic Algorithm for Approximation Betweenness Centrality Using Graph Coarsening. *Procedia Computer Science*, 66(2015), 83-92.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. ve Stein, C. (2001). *Introduction to Algorithms* (2. bs.). Cambridge, Massachusetts, London: McGraw-Hill Book Company.

Çelik, S. ve Akdamar, E. (2018). Büyük Veri ve Veri Görselleştirme. *Akademik Bakış Dergisi*, 65(2018), 253-264.

Doğan, K. ve Arslantekin, S. (2016). Büyük Veri: Önemi, Yapısı ve Günümüzdeki Durum. *Ankara Üniversitesi Dil ve Tarih-Coğrafya Fakültesi Dergisi*, 56(1), 15-36.

Kyrola, A. (2014). *Large-scale Graph Computation on Just a PC*. Doktora tezi, Carnegie Mellon University, Pittsburgh.

Lim, S. H., Lee, S., Ganesh, G., Brown, T. C. ve Sukumar, S. R. (2015). *2015 IEEE International Symposium on Performance Analysis of Systems and Software*, 42-51. doi: 10.1109/ISPASS.2015.7095783

Luaces, D., Viqueira, J.R.R., Cotos, J.M. ve Flores, J.C. (2021). Efficient Access Methods for Very Large Distributed Graph Databases. *Information Sciences*, 573, 65-81. https://doi.org/10.1016/j.ins.2021.05.047

Lv, X., Chen, S., Zheng, S., Luan, J. ve Guo, G. (2018). Understanding The Graph Databases and Power Grid Systems. *IOP Conference Series: Materials Science and Engineering*, 439(032110), 1-5. doi:10.1088/1757-899X/439/3/032110

Needham M. ve Hodler, A. E. (2019). *Graph Algorithms: Practical Examples in Apache Spark and Neo4j (1. bs.).* Gravenstein Highway North, Sebastopol: O'Reilly Media.

Papamanthou, C. (2004). *Depth First Search and Directed Acyclic Graphs: A Review for the Course Graph Algorithms*. Department of Computer Science University of Crete.

Ragab, M. (2020). Large Scale Querying and Processing for Property Graphs. *PhD Symposium*, 2572.

Rawat, D. S. ve Kashyap, N. K. (2017). Graph Database: A Complete GDBMS Survey. *International Journal for Innovative Research in Science & Technology*, 3(12), 217-226.

Robinson, I., Webber J. ve Eifrem E. (2015). *Graph Databases* (2. bs.). Gravenstein Highway North, Sebastopol: O'Reilly Media.

Xia, Y., Tanase, I. G., Nai, L., Tan, W., Liu, Y., Crawford, J. ve Lin C. Y. (2014). Explore Efficient Data Organization for Large Scale Graph Analytics and Storage. *International Conference on Big Data*, 942-951.

Yılmaz, M. (2009). Enformasyon ve Bilgi Kavramları Bağlamında Enformasyon Yönetimi ve Bilgi Yönetimi. *Ankara Üniversitesi Dil ve Tarih-Coğrafya Fakültesi Dergisi*, 49(1), 95-118.

*GAZİLİ OLMAK AYRICALIKTIR.*