EÜ FEN BİLİMLERİ ENSTİTÜSÜ

EGE ÜNİVERSİTESİ

YÜKSEK LİSANS TEZİ

Çizge Veritabanları Üzerinde İşlemler

Özge ERTEN

Tez Danışmanı: Prof. Dr. Aylin KANTARCI

Bilgisayar Mühendisliği Anabilim Dalı

Sunuş Tarihi : 29.09.2017

Bornova-İZMİR

2017

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

(YÜKSEK LİSANS TEZİ)

Çizge Veritabanları Üzerinde İşlemler

Özge ERTEN

Tez Danışmanı: Prof. Dr. Aylin KANTARCI

Bilgisayar Mühendisliği Anabilim Dalı

Sunuş Tarihi: 22.09.2017

Bornova-İZMİR

2017

Özge ERTEN tarafından Yüksek Lisans tezi olarak sunulan "Çizge Veritabanı Üzerinde İşlemler" başlıklı bu çalışma EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliği ile EÜ Fen Bilimleri Enstitüsü Eğitim ve Öğretim Yönergesi'nin ilgili hükümleri uyarınca tarafımızdan değerlendirilerek savunmaya değer bulunmuş ve 29.09.2017 tarihinde yapılan tez savunma sınavında aday oybirliği/oyçokluğu ile başarılı bulunmuştur.

Jüri Üyeleri:		**İmza**
Jüri Başkanı	: Prof. Dr. Aylin KANTARCI	.....
Raportör Üye	:Prof. Dr. Oğuz DİKENELLİ	.....
Üye	:Yrd. Doç. Dr. Korhan KARABULUT	.....

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

ETİK KURALLARA UYGUNLUK BEYANI

EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliğinin ilgili hükümleri uyarınca Yüksek Lisans Tezi olarak sunduğum "Çizge Veritabanları Üzerinde İşlemler" başlıklı bu tezin kendi çalışmam olduğunu, sunduğum tüm sonuç, doküman, bilgi ve belgeleri bizzat ve bu tez çalışması kapsamında elde ettiğimi, bu tez çalışmasıyla elde edilmeyen bütün bilgi ve yorumlara atıf yaptığımı ve bunları kaynaklar listesinde usulüne uygun olarak verdiğimi, tez çalışması ve yazımı sırasında patent ve telif haklarını ihlal edici bir davranışımın olmadığını, bu tezin herhangi bir bölümünü bu üniversite veya diğer bir üniversitede başka bir tez çalışması içinde sunmadığımı, bu tezin planlanmasından yazımına kadar bütün safhalarda bilimsel etik kurallarına uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda l edeceğimi beyan ederim.

29 / 09 / 2017

İmzası

Adı-Soyadı

Özge ERTEN

ÖZET
Çizge Veritabanları Üzerinde İşlemler

ERTEN, Özge

Yüksek Lisans Tezi, Bilgisayar Mühendisliği Anabilim Dalı
Tez Danışmanı: Prof. Dr. Aylin Kantarcı
Eylül 2017, 48 sayfa

Bu tezde yeni, gerçek hayatta uygulanabilir, Türkiye ve dünya için gelecek vadeden çizge büyük veri sistemlerinde çizge veritabanları konusunda inceleme ve uygulama yapmayı hedefledik. Gelecekte bu konunun popüler bir çalışma alanı olacağı kesindir. Bu konuda bilgi birikimi ve deneyim kazanmak tezin ana hedefleridir.

Titan veritabanı sunduğu imkânlar bakımından çizge veritabanı alanındaki diğer rakiplerinden ayrılır. Hızlı dolaşıma ve farklı makinelerde işleme izin vermesi en temel avantajlarıdır. Ancak yeni bir teknoloji oluşu sebebiyle piyasada bu veritabanı teknolojisiyle geliştirilmiş proje sayısı kısıtlıdır ve yetkinlik kazanmış kişi sayısı azdır. Çizge Veritabanı konusunun bu teknoloji ile birlikte öğrenilmesi ve bu konuda deneyim kazanılması önem teşkil etmektedir.

Tez kapsamında çizge veritabanları kullanılarak 2 ayrı veri tabanı tasarlanmış, gerçekleştirilmiş ve bu veri tabanları için çeşitli sorgular oluşturulmuştur. Çizge veritabanları ilişkisel veri tabanları ile bu örnekler üzerinden JOIN operasyonu sayısı göz önüne alınarak karşılaştırılmıştır. Ayrıca, Titan ve Neo4j`nin yapısal özellikleri (kullanım ve öğrenme kolaylığı, sorgu dili karşılaştırması) kıyaslanacaktır.

Anahtar Kelimeler: Büyük veri, NoSQL, Geleneksel veritabanı, Titan DB, Çizge veritabanı, Apache Cassandra

ABSTRACT
Operations On Graph Databases

ERTEN, Özge

MSc in Computer Eng.

Supervisor: Prof. Dr. Aylin Kantarcı

September 2017, 48 pages

In this thesis we aimed to examine and implement graph databases in big data systems which is a promising subject for Turkey and the world and it can be applied in real life. In the future it will be a popular field of study.The main aim of the thesis is to gain knowledge and experience in this field.

The Titan database is separated from other competitors in the graph database in terms of facilities. The most essential advantages of Titan database are fast traversal and allowing processing on different machines. However, due to the emergence of a new technology, the number of projects developed with this database technology and qualified people in the market is limited.It is important to learn graph database with this technology and gaining experience in this subject.

Within the scope of the thesis, 2 separate databases were designed and implemented using the graph databases, and various queries were made for these databases.Also, the graph databases are compared with the relational databases, taking into account the number of JOIN operations over these examples. In addition, the structural properties of Titan and Neo4j (ease of use and learning, query language comparison) will be compared.

Keywords: Big data, NoSQL, Relational Databases, Titan DB, Graph database, Apache Cassandra

TEŞEKKÜR

Bu tez çalışması boyunca gösterdiği her türlü destek ve yardımdan dolayı çok değerli hocam Sayın Prof. Dr. Aylin Kantarcı'ya ve Sayın Prof. Dr. Oğuz Dikenelli'ye içten dileklerimle teşekkür ediyorum. Ayrıca, hayatım boyunca benden değerli desteklerini esirgemeyen ve attığım her adımda yanımda olan aileme teşekkür etmeyi bir borç bilirim.

Eylül, 2017 Özge ERTEN

İÇİNDEKİLER

Sayfa

İÇİNDEKİLER (devam)

ŞEKİLLER DİZİNİ

Şekil Sayfa

ŞEKİLLER DİZİNİ (devam)

ÇİZELGELER DİZİNİ

Çizelge Sayfa

GİRİŞ

Gelişen teknoloji ile birlikte üretilen veri de her geçen gün artmaktadır. Sosyal medyada üretilen veriler; şirketlerin, müşteri geçmişlerini tuttuğu kayıtlar, sensörlerden devamlı akan veri, hastanelerde üretilen hasta verileri… Gibi birçok kaynaktan sürekli veri üretilmektedir. Geleneksel veritabanları, yapısal ve belirli bir büyüklüğe kadar olan verileri işleme ve saklama kabiliyetine sahiptir. Giderek artan saklama ve daha hızlı işlem, yapısal olmayan veriler, sabit boyutlu şema gibi problemler; büyük veri ve NoSQL terimlerinin ortaya çıkmasını sağlamıştır.

Büyük veri, geleneksel veritabanı sistemleri tarafından işlenemeyen veri setleri için kullanılır. Aşağıda açıklandığı üzere 4*V*, büyük verinin temelini oluşturmaktadır:

Günümüzdeki verinin yaklaşık %90'ı, son iki yılda üretilmiştir. Her gün 2,5 kentilyon byte veri üretilmektedir. *Volume*, verinin ölçeğini temsil eder. Gelen verinin büyüklüğü, Volume bileşeninin oluşmasına neden olmuştur.[11][12]

Velocity, verinin hızıdır. Her 60 saniyede 204.000.000 e-mail gönderilmektedir. Küresel internet trafiğinin 2018'de, 50.000 Gb/saniye olması öngörülmektedir. Analiz edilmek için gelen verinin hızı Velocity bileşeninin oluşmasına neden olmuştur.[11][12]

Veracity, verinin kesinliğini temsil eder. Amerika'da düşük veri kalitesinin her yıl 3,1 trilyon $ zarara uğrattığı tahmin edilmektedir. Artan veri miktarıyla birlikte gelen verinin kesinliği problemi Veracity bileşeninin oluşmasına neden olmuştur.[11][12]

Variety, verinin çeşitliliği için kullanılan bir kavramdır. Üretilen verinin yaklaşık %90'ı yapısal olmayan veridir (fotoğraflar, müşterin satın aldıklarının kayıtları...). Yapısal veriler genellikle sensörlerden gelir. Variety; verinin yapısal, yarı yapısal ve yapısal olmayan olarak üç farklı türü nedeniyle oluşmuş bir bileşendir.[11][12]

NoSQL, yani "Not only SQL", veritabanı yönetim sistemlerinin bir sınıfıdır ve sorgu dili olarak SQL'i kullanmaz. Bunun yerine NoSQL veritabanları kendi özel sorgu dillerini kullanmaktadır. Terabyte'lara ulaşan günlük işlem hızına izin veren, dağıtık mimariye izin veren NoSQL teknolojileri ile büyük veriyi işlemek, analiz etmek ve saklamak mümkündür.

Kullanılacak teknolojilerin seçimi alınacak verimin en üst seviyede olması için gereklidir. Bir şirketin ya da kurumun sakladığı verinin türü, boyutu, ilişkiler, saklama yapısı… Gibi birçok unsur göz önüne alınmalıdır. Key-value store, column oriented, document store ve çizge olmak üzere dört çeşit NoSQL tipi ihtiyaç duyulan özelliklere göre seçilebilir. [14][15]

Key-value store depolama sistemleri, oldukça basit fakat etkin ve güçlü modellerdir. Temel olarak açıklamak gerekirse, anahtar ve değerlerden oluşan birleştirici dizilerdir. Anahtar genellikle basit bir objeyken değer daha karmaşık veri yapılarına izin verir; bir liste, bir dizi seti, hash… Key-value store'lar, anahtar kısmı bir indeksleme mekanizması olarak kullanmasıyla hash tablolarıyla benzerlik gösterir. Bu key-value store'u, geleneksel veritabanlarından daha hızlı yapar. Çünkü veri modeli basittir. Bir kullanıcının oturum veya alışveriş sepeti kayıtları tutulacağı durumlarda, favori ürün gibi detayların öğrenilmesi istendiğinde bu veritabanı türü kullanılabilir. Amazon DynamoDB ve RIAK key-value store veritabanlarıyla çalışan popüler teknolojilerdir. [14][15]

Column store, saf ilişkisel sütun veritabanlarının aksine hibrit satır/sütun veritabaıdır. Sütundan sütuna saklama konseptini paylaşmasına rağmen veriler tabloda değil, kitlesel dağıtık bir mimaride saklanır. Her anahtar, bir veya birden fazla attribute (sütun) ile ilişkilidir. Saklama şekli sayesinde istenen veriler daha az I/O etkinliğiyle hızlı bir şekilde toplanır. [14][15]

Document store, verilerin bir çeşit dokümanda tutulduğu veritabanını temsil eder. Bu veritabanı; saklanacak verinin belirli büyüklüğe ihtiyaç duymayacağı, bunun yerine özel karakteristikli bir dokümanda saklanması gereken uygulamalar için kullanılır. Çok fazla ilişkinin ve normalizasyonun olduğu durumlarda bu veritabanı uygun değildir. MongoDB ve CouchDB, popüler document store teknolojileridir. [14][15]

Çizge veritabanları ise verileri bir çeşit çizgede saklar. *Düğüm* (node) ve *Kenar* (edge) bileşenleri objeler ve ilişkileri temsil eder. Ayrıca düğümlerle ilişkili *özellikler* (property) bulunur. Bu, indeks gerektirmeyen bitişiklilik tekniğidir. Her düğüm, bitişiklilik düğümünlerine işaret eden işaretlere (pointer) sahiptir. Bu sayede milyonlarca kayıt gezilebilir. Çizge veritabanı veriler arası ilişkiler üzerinde odaklanır. Yarı yapılandırılmış verilerin etkin bir şekilde saklandığı, az şemalı bir yapı sağlar. Sosyal medya uygulamaları, biyoinformatik, içerik yönetimi, güvenlik ve erişim kontrolü gibi pek çok alanda çizge veritabanı kullanılabilir. Neo4j ve TitanDB, popüler çizge veritabanı uygulamalarıdır. [14][15]

Günlük hayatta pek çok alan, sürekli büyüyen bir veri kümesini barındırır. Bu veriden sorgular, yöntemler ve analizlerle bilgiye ulaşmak zaman, maliyet, depolama… Gibi birçok özellik yönünden fayda sağlar. Büyük veriden işlenerek elde edilen bilgi, değer açısından önemlidir. Sonuçtaki veriye göre satış stratejisi geliştirilebilir, daha doğru kararlar verilebilir. Ancak büyük veri analizinde birtakım sorunlarla karşılaşılmaktadır: [13]

Mevcut teknolojiler yetersiz, girdi/çıktı erişimi yavaş ve veri çeşitliliği fazladır. Bu sorunları çözmek saklanan verinin gereksinimleri iyi bir şekilde anlaşılmalı ve en uygun teknolojiler seçilmelidir. Ayrıca, verinin tekrar ettiği

durumlar, eksik veya hatalı kısımlar veri madenciliği uygulamaları ile tespit edilmelidir. [13]

Bilgi keşfi ve gösterimi büyük veri alanında ana problemlerden biridir. Büyük bir sorun olmasının bir nedeni de arşivleme, yönetim... Gibi alt konular da içermesidir. Bilgi keşfi için birçok araç bulunmaktadır. Gerçek dünya problemleriyle baş edebilmek için bunlar hibritleştirilerek yeni araçlar oluşturulabilir. Büyük verileri analiz etmek daha fazla işlemsel karmaşıklık gerektirir. Ana sorunlar, veri setindeki tutarsızlıklar ve kesin olmayan gösterimlerdir. Büyük veri boyutu, CPU hızından daha hızlı ölçeklenmektedir. Bu sebeple işlemci teknolojisi, gömülü ve sayısı artan çekirdekler yönünde gelişim göstermektedir. Ayrıca bu gelişim paralel hesaplamanında gelişimini sağlamıştır. Sosyal ağlar, navigasyon gibi gerçek zamanlı uygulamalar paralel hesaplamaya ihtiyaç duyar. [13]

Büyük veri görselleştirme araçları, büyük ve karmaşık verileri resim, grafik... Vb. şekillerde görselleştirme yeteneği sahiptir. Ancak günümüz araçlarının çoğu tepki süresi, ölçeklenebilirlik ve performans konularında eksik kalmaktadır. [13]

Büyük veriyi analiz etmek için çok fazla sayıdaki veri ilişkilendiriliyor, analiz ediliyor ve anlamlı desenler için veri madenciliği yapılıyor. Elde edilen değerli bilginin güvenliği önemli bir unsurdur. Çoğu büyük veri uygulaması; ağ ölçeği, cihazların çeşitliliği, gerçek zamanlı güvenliğin görüntülenmesi, ihlal sistemi eksikliği gibi güvenlik ölçümü konularıyla yüzleşmektedir. [13]

Bu konuda güvenliği arttırmak için yetkilendirme, kimlik doğrulama ve şifreleme gibi teknikler kullanılabilir. Ayrıca çok seviyeli güvenlik politikası geliştirimine odaklanmak gerekmektedir.[13]

Görüldüğü gibi çizge veri tabanları bilişim dünyasında birçok sektörü etkilemektedir. Bu veritabanları büyük veri uygulamaları yaygınlaştıkça daha çok uzman tarafından öğrenilecek ve kullanılacaktır. Bu tezde hedeflenen bir çizge veri tabanı oluşturma, sorgulama gibi konularda deneyim oluşturmaktır. İlerleyen bölümlerde çizge veritabanları, bunların kullanımları açıklandıktan sonra 2 çizge veritabanı örneği sunulacaktır. Bu çizge veritabanlarının oluşturulması ve sorgu detayları okuyucu ile paylaşılacaktır. Çizge veritabanlarının getirdiği avantajlar da yeri geldikçe açıklanacaktır.

BÖLÜM 1. KULLANILACAK TEKNOLOJİ, PLATFORM VE PROJENİN YOL HARİTASI

1.1. Arama Kriterleri

Büyük veri ve operasyonları hakkında veri ve yapılan çalışmalara erişim için şu anahtar kelimeler ve kombinasyonları kullanılarak bir arama stratejisi geliştirilmiştir: Patient records, health ontology, big data, çizge veritabanı, Titan graph database, big data analysis, healthcare applications in big data, medical applications in big data, medical analysis applications, medical ontology, radiology ontology

Arama yapılırken temel olarak kullanılan internet siteleri şöyledir: Google Scholar, ieeexplore.ieee.org, PubMed.

1.2. Uygun Yazılımlar Ve Ortamlar

Tez danışmanlarımın deneyimi ve bilgi birikimiyle kullanılacak teknolojilerin Windows ortamında ve Java araçları desteğiyle ile desteklenmesine karar verildi. Bu kararı verirken belirtilen teknolojilerin güvenilir olması, yapılacak proje teknolojilerini birleştirici oluşu ve birçok platformdaki hata desteği ve yardım forumlarının bulunması etkili oldu.

Veritabanı seçiminde geliştirilen yapıya uygun olarak saklanacak olan büyük miktardaki hastane girdisi verilerinin ölçeklenebilir olması açısından çizge veritabanında saklanmasına karar verilmiştir. Piyasada çok çeşitli NoSQL veri saklama teknolojileri bulunmaktadır. Önceki bölümlerde anlatılan key-value store, column store, document store bu veritabanlarına en temel örneklerdir. Ancak bu veritabanlarından hiçbiri veriyi ve ilişkilerini saklamak konusunda çizge veritabanı kadar performanslı değildir. Çizge veritabanının belki de en büyük artısı bağlantıları ve ilişkilerini birinci sınıf entity'ler olarak saklama yeteneğidir. Çizge veritabanı CREATE, UPDATE, READ ve DELETE operasyonlarını bir çizge modeli üzerinde çalıştırır. Bu veritabanları işlemsel sistemler (OLTP) üzerinde çalışmak üzere inşa edilmiştir. Diğer veritabanlarının aksine çizge veritabanı ilişkileri birinci önceliğe alır. Bunun anlamı şudur: Uygulamanın, veri bağlantıları için foreign key'ler ile ya da MapReduce gibi işlemlerle çıkarım yapmasına gerek yoktur.

Sorgu performansı ve tepki süresi çoğu firma için öncelikli konulardır. Online işlemsel sistemler kullanıcılarına milisaniyeler içerisinde cevap verebiliyorsa başarılı olarak nitelendirilir. İlişkisel yapıdaki sistemlerde veri seti

büyüdükçe JOIN sorunu büyür ve performans düşer. Index-free mantığını kullanan çizge veritabanı karmaşık JOIN operasyonlarını hızlı çizge çaprazlamalarına dönüştürmüştür. Bu sayede milisaniyeler içerisinde istenen verim alınabilir.[16]

Var olan popüler çizge veritabanı saklama teknolojilerinden Neo4j yalnızca tek makine üzerinde çalışabildiğinden ölçeklenebilir değildir. Titan büyük miktarlarda veri barındıran çizgeleri etkin saklamayı ve sorgulamayı amaçlayan ölçeklenebilir bir çizge veritabanıdır. Titan verileri fiziksel olarak depolarken arka-uç kalıcılık katmanı kısmında seçimli olarak farklı ihtiyaçlara göre Cassandra, HBase veya BerkeleyDB veritabanlarından birini kullanmaktadır. Bu arka-uç veritabanlarından Cassandra kullanıldığında performanstan ödün verilmeyip, aynı zamanda ölçeklenebilirlik (scalability) ve yüksek uygunluk (high availability) sağlanabilmektedir. Bu sebeple arka planda Cassandra ile konuşan Titan veritabanı sistemi projede kullanılacaktır.

1.3. Kullanılacak Teknolojilerin Öğrenilmesi

Çizge veritabanlarının genel özellikleri, yapısı ve kullanım alanlarını öğrenmek için "Graph Databases by Ian Robinson, Jim Webber and Emil Eifrem" ve "Big Data For Dummies" temel NoSQL anlatımı yapan kitapları incelendi. Bu aşamadan sonra, Big Data isimli doktora dersini alarak ve bu ders için yapılan projede NoSQL sistemler ve kullanılan veritabanı teknolojilerinin anlatımını içeren bir proje ve tez danışmanı tarafından istenen NoSQL konusunda bir sunum ile bu alana giriş yaptım.

Proje uygulaması geliştirme adımı için gereken bilgi birikimin bir kısmı için internetteki forumlar, sayfalar, program örnekleri ve kullanılacak teknolojinin dokümantasyonunu içeren sitelerden yararlanılmıştır.

Ayrıca yapılan benzer alanlara yönelik araştırma ve projeler incelenmiştir:

"Big Data In Health Care: Using Analytics To Identify And Manage High-Risk And High-Cost Patients" adlı çalışmada Amerika'da sağlık alanında büyük veri ve operasyonların etkileri anlatılmıştır. Bu çalışmaya göre büyük veri kullanımı bazı sağlık alanlarındaki giderleri azaltmış ve kullanılan analiz yöntemleriyle yeni bakış açıları kazanılmıştır. Söz konusu araştırmada kullanılan sorular şöyledir. [17]

Q1: Hangi hastaların yüksek riskli ya da yüksek giderli olduğunu tahminlemek için nasıl bir yaklaşım uygulanmalı?

Q2: Hangi yeni ölçüm kaynaklarının birlikte kullanımı tahminlemeleri geliştirebilir?

Q3: Hangi hastaların tahminlemeler kullanılarak yapılacak müdahalelerle daha fazla yarar sağlayabilir ve hangi spesifik müdahaleler sağlığı daha fazla geliştirir?

Q4: Tahminleme modelleri oluştururken hangi durum çıktılarının giderleri düşük riskli grubuna alınmalı? [17]

Bu soruların ana fikrinden hareketle, tüm hastaların analiz edilerek hangi hastaların yüksek riskli olduğunun belirlenmesini sağlayan bir algoritma geliştirilmelidir. Ayrıca bu algoritma düşük risk grubu hastaları da tespit etmelidir. Böylece geliştirilen algoritma ile gereksiz yere her iki haftada bir hastane randevusu alan hasta ve her iki haftada bir hastaneye gitmesi gereken önemli sağlık sorunu bulunan hasta arasında ayrım yapılabilir. [17]

Araştırmanın öne sürdüğü durum şudur; Amerika'da sağlık giderlerinin %50'lik kısmı, %5'lik hasta kitlesi tarafından oluşturulmaktadır. Bu hastalar tanımlandığında giderlerin azaltılması sağlanır. Bu fikri ispatlamak için efektif analiz metodları, uygulanacak metodların benzer bir populasyona uygulanması, hastalar için davranışsal sağlık sorunlarının belirlenmesi unsurları deney seti için belirlenmiş olmalıdır.[17]

"Semantic Description of Liver CT Images: An Ontological Approach" isimli çalışmada, radyoloji verileri kayıtlarını tutmak ve bunlar arası ilişkileri belirlemek için mantıksal bir modele duyulan ihtiyaç nedeniyle geliştirilen ONLIRA ontolojisi tanıtılmaktadır. ONLIRA, genel olarak benzer hasta durumlarını belirlemek ve derecelendirmek üzere oluşturulan akıllı uygulamaları desteklemek için geliştirilmiştir. [18]

Çalışmada öne sürülen durum şudur; karaciğere ait radyolojik görüntü verilerini, yapısal ve standardize tutmak fayda sağlar. Bu verilerin ontolojik modellenmesi daha efektiftir. Bu fikri ispatlamak için radyoloji verileri incelenerek modellenmiş ontoloji**,** işlem performansı işlem zamanı ve saklanan verilerin bakımı gibi sayılarla ifade edilebilinen deney setleri kullanılabilir.[18]

"Managing, Analysing, and Integrating Big Data in Medical Bioinformatics: Open Problems and Future Perspectives" adlı çalışmada sağlık alanında gelen büyük veririnin analiz edilmesiyle hastalıkların daha iyi anlaşılabilmesi, kişisel tedaviler, yeni tedavi ve tedavi yöntemleri geliştirilebileceği anlatılmıştır. Bu bilgilere ulaşabilmek için çok büyük miktarda veriyle baş edilebilmeli ve yeni paradigmalar geliştirilmelidir. Çalışma biyoinformatik alanında kullanılan şu ontolojileri tanıtmaktadır: [19]

Gene Ontology,biyomoleküler alanda en çok kullanılan çok seviyeli ontolojidir. Genomlar ve ilgili bilgileri; moleküler fonksiyonlarına göre ayıran genleri ve proteinleri karakterize etmek için uygun, çizge tabanlı bir hiyerarşik yapıda tutar. [19]

KEGG ontology (KOnt), tüm organizmaların genlerinin açıklamasına dayalı bir yol sağlar.

Brenda Tissue Ontology (BTO), insan dokularının açıklamasını içerir.

Cell Ontology (CL), hücre tipleri hakkında kapsamlı bir organizasyon sağlar.

Disease ontology (DOID), meme kanseri patolojisinin diğer hastalık türleriyle sınıflandırılması üzerinde durur.

Protein Ontology (PRO), bir genin çoklu protein sınıfları için farklı protein evrim sınıfları tanımlar.

Medical Subject Headings thesaurus (MESH); biyomedikal ve sağlıkla ilgili bilgileri indeksleyebilecek, hiyerarşik kelime haznesidir.

Protein structure classification (CATH), protein yapılarını sınıflandıran yapısal kelime haznesidir. [19]

Ayrıca, standart ve doğru bir şekilde büyük veri analizi için kullanılması gereken büyük veri mimarisi, büyük veri analizinde kullanılacak hesaplamalar, veri erişimi ve güvenliği hakkında çeşitli yöntemleri tanımlamaktadır.[19]

"A Survey on Big Data Analytics: Challenges, Open Research Issues and Tools" adlı çalışma büyük veri analitiği ve karşılaşılan zorlukları açıklamaktadır. Verinin önemini açıklayan en iyi durum, kritik karar süreçlerine katkılarıdır. Belirli yöntemlerle işlenen veriden bilgi çıkarımı yapılır ve değerli bilgiye ulaşılır. Büyük veri analizinde karşılaşılan 4 ana zorluk şöyledir; [20]

1. Veri Depolama Ve Analiz: Mevcut teknolojiler yetersiz, input/output erişimi yavaş,veri çeşitliliği, Data reduction ve Data selection yani gelen verilerin eksik, tekrarlayan, hatalı olup olmadığı problemli. [20]

2. Bilgi Keşfi ve Hesaplama Karmaşıklığı: Bilgi keşfi ve gösterimi büyük veri alanında ana problemlerden biridir. Büyük bir sorun olmasının bir nedeni de arşivleme, yönetim... Gibi alt konular da içermesidir. Bilgi keşfi için birçok araç bulunmaktadır. Gerçek dünya problemleriyle baş edebilmek için bunlar hibritleştirilerek yeni araçlar oluşturulabilir. [20]

Büyük verileri analiz etmek daha fazla işlemsel karmaşıklık gerektirir. Ana sorunlar, veri setindeki tutarsızlıklar ve kesin olmayan gösterimlerdir.

3. Ölçeklenebilirlik ve veri gösterimi: Büyük veri boyutu, CPU hızından daha hızlı ölçeklenmektedir. Bu sebeple işlemci teknolojisi, gömülü ve sayısı artan çekirdekler yönünde gelişim göstermektedir. Ayrıca bu gelişim paralel hesaplamanın da gelişimini sağlamıştır. Sosyal ağlar, navigasyon gibi gerçek zamanlı uygulamalar paralel hesaplamaya ihtiyaç duyar. [20]

Büyük veri görselleştirme araçları, büyük ve karmaşık verileri resim, grafik... Vb. şekillerde görselleştirme yeteneği sahiptir. Ancak günümüz araçlarının çoğu tepki süresi, ölçeklenebilirlik ve performans konularında eksik kalmaktadır. [20]

4. Bilgi Güvenliği: Büyük veriyi analiz etmek için çok fazla sayıdaki veri ilişkilendiriliyor, analiz ediliyor ve anlamlı desenler için veri madenciliği yapılıyor. Elde edilen değerli bilginin güvenliği önemli bir unsur. Çoğu büyük veri uygulaması; ağ ölçeği, cihazların çeşitliliği, gerçek zamanlı güvenliğin görüntülenmesi, ihlal sistemi eksikliği gibi güvenlik ölçümü konularıyla yüzleşmektedir. [20]

Bu konuda güvenliği arttırmak için yetkilendirme, kimlik doğrulama ve şifreleme gibi teknikler kullanılabilir. Ayrıca çok seviyeli güvenlik politikası geliştirimine odaklanmak gerekmektedir.[20]

"The Inevitable Application of Big Data to Health Care" adlı çalışma kısaca şöyledir: Büyük veri astronomi, arama motorları hatta politikada büyük ölçüde kullanılmaktadır. Örneğin, Google arama motoru, alınan anahtar kelimelere göre internette bulunan tüm veriyi tarayarak kullanıcıya en uygun sayfaları getirmektedir. Bu nedenle, bu ileri teknolojinin sağlık alanında kullanılması sağlık hizmetlerinin daha kaliteli ve etkili karşılanmasına büyük fayda sağlayacağı düşünülmektedir. İlk olarak, toplanan veri yeni bilgiler ya da bulgular öğrenilmesine katkı sağlayabilir. Çünkü bir hastanın bilgilerine direkt olarak ulaşmak yasaktır, ancak bütün hastaların bilgisi toplanıp genel bilgilere ulaşmak sorun teşkil etmemektedir. İkinci olarak, toplanan bilgilerin yayılması sağlanabilir. Örneğin, küçük bir kliniğe gelen hasta sayısı büyük bir hastaneye gelen sayısından çok azdır. Bu nedenle, yeni bulgularla karşılaşılma olasılığı düşüktür. Ancak, büyük veri sayesinde oluşturulan tüm bilgiler anonim şekilde diğer kliniklere yardımcı olacaktır. Üçüncü olarak, kullanılan kişisel tedavi girişimleri analiz edilerek klinik tedavilerinde kullanılacak hale dönüştürülebilir. Son olarak, toplanan veriler ve oluşturulan sonuçlar hastalara ulaştırılarak onların daha çok bilgi sahibi olmaları, hatta gerektiğinde kendileri tedavi edecek hale getirilmesi sağlanabilir. Bu nedenle, büyük verinin sağlık hizmetlerinde kullanımı, toplanan bilgilerin ileride sürekli kullanılacak hale getirilerek sağlık sorunlarını daha etkili ve kaliteli bir şekilde çözmemizi sağlayacaktır.[22]

"Big Data Analytics for Healthcare" adlı çalışmanın amacı veri madenciliği ve medikal topluluklar arasında köprü kurarak bir disiplinler arası çalışma ortaya koymaktır. Buna göre, karmaşık ve büyük veri setlerini yönetmek geleneksel veritabanı yönetim sistemleri ile işlenmesi için efektif değildir. Büyük veri bu tür verileri yönetmek, işlemek… Gibi uygulamaları gerçekleştirmek üzere geliştirilmiştir. Sağlık alanında büyük veri; karmaşık ve heterojen hasta kaynakları, yapısal olmayan klinik notların anlaşılması, çok büyük hacimli görsel medikal sonuçları etkili bir şekilde saklama ve önemli bilgiye ulaşılması, genom

verilerinin analizi, hastadan anlık veri toplanması ve çeşitli uyarı sistemleri geliştirilmesi gibi konularda kullanılmaktadır. Doğru verilerden yapılan doğru analizin medikal alanda çok büyük avantajları vardır. Erken teşhis, daha düşük sağlık harcamaları, hasta-doktor uyarı sistemleri bunlardan bazılarıdır. Çalışmada verilen örnekler, büyük verinin medikal alandaki kazanımlarını göstermektedir. Giderek artan sağlık verisi için büyük veri doğru bir yöndür. Sağlık verisinin büyüklüğü, büyük veri için artacak talebi göstermektedir. Büyük veri tabanlı karmaşıklıkları çözümlemek hastalar için doğru zamanda doğru kararların verilmesini sağlayacaktır.[23]

Benchmarking yani kıyaslama, sistemlerin belirli metriklerle (hız, doğruluk, uygunluk, tepki süresi...) ve belirli iş yükleri uygulanarak incelenmesi, analizi ve gözlemlenmesidir.

Metrik seçimi, değerlendirme tekniği temel alınarak yapılır. Üç temel değerlendirme tekniği bulunur. Bunlar; analitik modelleme, simülasyon ve ölçümlerdir.

Kriter	**Analitik Modelleme**	**Ölçüm**	**Simülasyon**
Aşama	**Herhangi bir aşama**	**Herhangi bir aşama**	**İlk Örnek aşaması**
Gereken Zaman	**Az**	**Orta**	**Değişken**
Doğruluk	**Düşük**	**Orta**	**Değişken**
Maliyet	**Düşük**	**Orta**	**Yüksek**

Tablo 1-1: Metrik seçimi

İş yükü kıyaslanacak sistemlerle uyumlu olmalıdır. Tekrarlanabilirlik önemlidir. Gerçek kullanıcı davranışları gözlemlenmeli ve bu temel alınarak tekrarlanabilir bir iş yükü oluşturulmalıdır.[26]

"Benchmarking graph databases on the problem of community detection" adlı çalışmanın çıkış noktası şudur; her geçen gün hızla büyüyen çizge verisini, linked data ve Online Social Networks (OSNs)'i saklamak, yönetmek ve analiz etmek çok kritik bir problemdir. Bu çalışma OSN madenciliği use case senaryolarından ilham alınarak çizge veritabanları bir dizi iş yükü ile kıyaslanmıştır. Popüler çizge veritabanlarından Titan DB, Neo4j ve Orient DB arasında, topluluk algılama sorununa en iyi çözüm getiren veritabanı için kapsamlı karşılaştırmalar yapılmış ve bunlar değerlendirilmiştir.[27]

Bu survey'de önerilen kıyaslama dört farklı iş yükünden oluşmaktadır:

Kümeleme, Büyük Ekleme, Tekli Yerleştirme ve Sorgu iş yükleri. Her bir iş yükü, çizge veritabanlarındaki genel operasyonları simüle etmek üzere dizayn edilmiştir. Çalışmanın temel iş yükü Kümeleme İş yüküdür ancak, kapsamlı bir karşılaştırma yapabilmek için ek iş yüklerinden de yararlanılmıştır. [27]

Şimdiye kadar kullanılan topluluk algılama algoritmaları, çizgeyi saklamak ve gereken işlemleri gerçekleştirmek için main memory'i kullanıyordu. Main memory'nin bu iş için kullanılması hızlı bir seçenek olsa da, büyük veri uygulamaları için en önemli unsurlardan biri olan güvenilir yönetim konusunda zayıf kalmaktadır. Bu çalışma daha çok Louvain metodunun üç veritabanında çalıştırılması üzerinde durmuştur. İlk çalıştırmada gereken tüm değerler veritabanlarından direk olarak okunmuştur. Bunun sonucunda görülmüştür ki memory'le kıyasla herhangi bir veritabanına erişim çok yavaştır. Böylece çalışmada cache teknikleri kullanılmasında karar kılınmıştır. İlk örnekte fonksiyonlar tarafından getirilecek veriler direk olarak varitabanından ve cache'den çekilerek bir değerlendirme yapılmıştır. [27]

Kümeleme iş yükünün yanında şu destekleyici iş yükleri de kullanılmıştır:

Büyük ekleme iş yükü için bir veritabanı yaratılmış, büyük veriye uygun halde ayarlanmış ve belirli bir veris etiyle yüklenmiştir. Bu örnekte çizge veritabanının yaratılma süresi ölçülmüştür.

Tekli ekleme iş yükü için bir veritabanı yaratılmış ve bir veri setiyle yüklenmiştir. Node'lar ve edge'ler arttıtımlı olarak eklenerek çizge yaratılmıştır. Bu örnekte 1000 edge ve 1000 node içeren blokların eklenme zamanı ölçümü yapılmıştır.

Sorgulama iş yükü için üç genel sorgu çalıştırılmıştır:

FindNeighbours (FN): Tüm node'ların komşusunu bulan sorgu

FindAdjacentNodes (FA): Tüm edge'lerin bitişik node'unu bulan sorgu

FindShortestPath (FS): İlk node'la rasgele seçilen 100 node arasındaki en kısa yolu bulan sorgu.

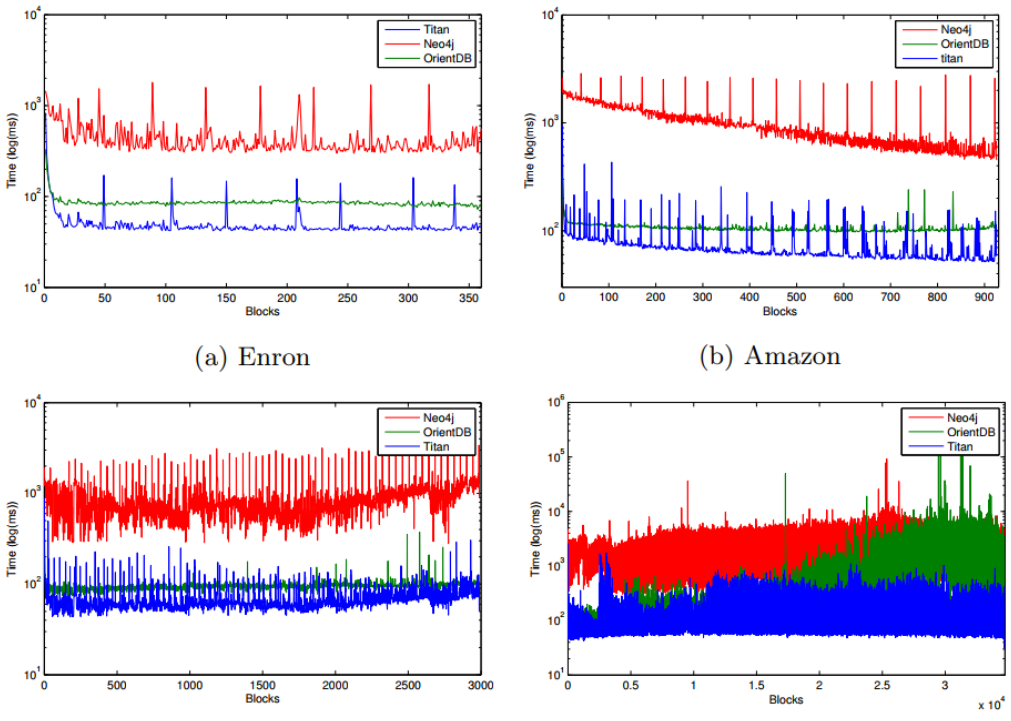
Bu sorguların her biri için zaman ölçüleri yapılmıştır.

Kullanılacak veri seti, kıyaslamada mantıklı ve karakteristik sonuçlara ulaşmak açısından önem taşır. Farklı karmaşıklıkta ve büyüklükte veri setleriyle kıyaslamalar yapmak bir veritabanını test etmek için gereklidir. Bu çalışmada sentetik ve gerçek veri setleri kullanılarak değerlendirmeler yapılmıştır. [27]

Graph	Workload	Titan	OrientDB	Neo4j
EN	MIW	9.36	62.77	**6.77**
AM	MIW	34.00	97.00	**10.61**
YT	MIW	104.27	252.15	**24.69**
LJ	MIW	663.03	9416.74	**349.55**
EN	QW-FN	1.87	**0.56**	0.95
AM	QW-FN	6.47	3.50	**1.85**
YT	QW-FN	20.71	9.34	**4.51**
LJ	QW-FN	213.41	303.09	**47.07**
EN	QW-FA	3.78	0.71	**0.16**
AM	QW-FA	13.77	2.30	**0.36**
YT	QW-FA	42.82	6.15	**1.46**
LJ	QW-FA	460.25	518.12	**47.07**
EN	QW-FS	1.63	3.09	**0.16**
AM	QW-FS	0.12	83.29	**0.302**
YT	QW-FS	24.87	23.47	**0.08**
LJ	QW-FS	123.50	86.87	**18.13**

Tablo 1-2: Büyük ekleme ve sorgulama iş yükü sonuçları

Verilen tabloda gerçek veri setleri kullanarak ulaşılmış sonuçlar görülmektedir. Buradan görülmektedir ki Neo4j ,büyük ölçekteki veriyi rakiplerinden daha başarılı bir şekilde yönetebilmektedir. Ayrıca Titan DB'de efektif bir alternatiftir. Oriend DB, rakiplerine kıyasla yavaş kalmıştır. FN algoritması için Orient DB biraz daha hızlı olsa da genel olarak Sorgu İş yükünde en etkin veritabanı Neo4j olmuştur. Tekli ekleme iş yükü göz önüne alındığında Titan rakiplerine kıyasla daha efektif sonuçlar üretmiştir. [27]



(a) Enron (b) Amazon

(c) Youtube (d) Livejournal

Tablo 1-3: kümeleme iş yükü

Tabloda kümeleme iş yükü için veritabanları kıyaslama sonuçları verilmiştir. 1000'den az node'lu çizgelerde Titan hızlıyken daha çok node içeren çizgelerde Neo4j öne geçmiştir. Ayrıca bu tabloda cache boyutunun arttıkça sürenin kısaldığı gözlemlenmiştir. [27]

Deneyler göstermiştir ki büyük bir şekilde artan verisetleri ile çalışıldığında en iyi sonuç Neo4j ile alınmaktadır. Tekli ekleme için Titan DB en etkili veritabanı iken, Orient DB Louvin metodunu uygulanırken kullanılacak en etkin veritabanı olmuştur. [27]

"Benchmarking traversal operations over graph databases" adlı çalışmada farklı veritabanlarının performansları kıyaslanmıştır. Traversal yani gezinme operasyonlar üzerinde yoğunlaşılmış ve bu kıyaslamaya uygun bir benchmark tasarlanmıştır. [28]

Çalışmaya göre çizge gezinme kıyaslaması şu iki özelliği test etmelidir:

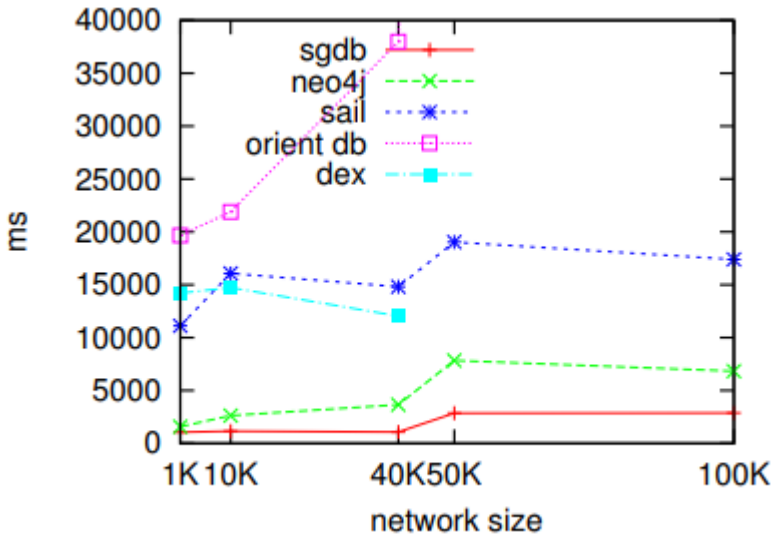
Lokal gezinmeler yani bir veya birkaç vertex'den başlanarak k. komşunun keşfi ve tüm çizgenin gezinimi (topluluk tespiti....). Ayrıca tüm çizgenin ana bellekten önbelleğe alınamadığı durumda test edilmelidir. [28]

Çalışmada bellek ile kısıtlı ortamlarda çizge veritabanlarının nasıl çalıştığı sorusuna, farklı boyutlarda verisetleriyle birlikte yanıt aranmıştır. Bu çalışmanın amacı önbelleğe alınamayacak büyüklükte çizgelerin bellekte nasıl yönetilebildiğini test etmektir. [28]

Benchmarking sonuçlarında dikkate alınan en önemli unsur adil olmaktır. Metadoloji önyargılar içermemelidir. Bu problem için Blueprint tüm çizge veritabanlarına bir interface olarak adapte edilmiştir. [28]

Kıyaslamalar Neo4J , DEX, Orient DB , Native RDF repository (NativeSail) ve bir araştırma prototipi olan SGDB veritabanları ile yapılmıştır.

İlk kıyaslamada lokal gezinim testi yapılmıştır. Bunun için verilen vertex'ten 3 atlama ile rastgele 10.000 vertex üzerinden çalıştırılmıştır. Veri seti 1000, 10000, 40000, 50000 ve 100000 vertix; ve daha büyük veri setleri 200, 400, 800 bin ve 1 milyon vertex veritabanlarına yüklenerek denemeler yapılmıştır. [28]



Tablo 1-4: 3 atlama ile yapılan gezinimin sonuçları

Tabloda görüldüğü üzere, 10.000 rasgele vertex ile 3 atlamalı gezinim yaptırılan 5 farklı veritabanından en etkin sonuçların alındığı veritabanı SGDB olmuştur. [28]

1.4. Yazılım Geliştirimi

Öncelikle, büyük verinin çizge veritabanında tutulmasının avantaj ve dezavantajları neler? Sorusundan yola çıkılarak basit yapıda örnek veri setleriyle, biri tıp alanında diğeri kişiler ve ilişkilerini kapsayan nitelikte olmak üzere iki adet veritabanı geliştirilecektir. Ayrıca, aynı veritabanı ilişkisel bir mantıkla gerçekleştirilseydi performans, karmaşıklık ve işlevsellik bakımından nasıl olurdu? Sorusuna yanıt aranacaktır. Bu nedenle, tasarlayıp geliştirdiğimiz veritabanları için hazırlanan sorgular, SQL diline dönüştürülecek ve join sayıları göz önüne alınarak performans kazancı hakkında çıkarımlar yapılacaktır.

BÖLÜM 2. UYGULAMA

Titan veritabanında büyük veri operasyonları konusunda gerekli deneyim ve öğrenme için uygulama geliştirimi yapılmıştır. Bu amaçla yapılan çalışmada 2 temel çizge veritabanı oluşturulmuş ve basitten karmaşığa bir dizi çalışma gerçekleştirilmiştir. Hasta Girdi Sistemi ve Person veritabanı çalışmaları altta açıklandığı gibidir:

2.1. Hasta Girdi Sistemi Veritabanı

Hasta girdi sistemi için çizge veritabanı ile modellenen bir sistem tasarlanmıştır. Bu sistemin ayrıntıları şöyledir:

Girdisi yapılacak hasta ve muayene bilgisinin saklanması için kullanılacak Apache Cassandra, “local server” modunda çalışacak olup Titan veritabanıyla “localhost” soketler aracılığıyla konuşacaktır. Bu bağlantıyı sağlamak için öncelikle Apache Cassandra bilgisayara kurulmuştur. Ardından Titan JAVA API, Eclipse IDE ile kullanılmıştır. Bunun için Java dilinde yazılmış bağlantı kodları şöyledir:

```
private static  TitanGraph titanGraph;

initializeTitanGraph() {

        configuration = new BaseConfiguration();

        configuration.setProperty("storage.backend", "cassandra");

        configuration.setProperty("storage.hostname", "127.0.0.1");

        titanGraph = TitanFactory.open(configuration);


titanGraph.makePropertyKey("HASTA_TC_NO").dataType(String.class).make();

            }
```

Burada gereken konfigürasyon ayarlamaları setProperty() metodu aracılığla verilmiş ve titanGraph adında bir Çizge veritabanı oluşturulmuştur. makePropertyKey() ile “HASTA_TC_NO”, property key olarak tanımlanmıştır.

Bu tanımlamayla key-value pair yapısı oluşturulmuştur. “HASTA_TC_NO”:”13” örneğinde “13”, “HASTA_TC_NO” key’inin value’sudur.

Hasta-Muayene bilgileri JSON formatında bir .txt dosyasında tutulmaktadır. Buradan alınan kayıtlar addhasta() ve addmuayene() metodlarında String formatına dönüştürülerek her bir hasta için bir vertex (örnek kod: Vertex Hasta = titanGraph.addVertex(null)) ve her bir muayene için bir vertex (örnek kod: Vertex Muayene_node = titanGraph.addVertex(null) oluşturulmuştur. Bir Hasta vertex için oluşturulan property’ler:

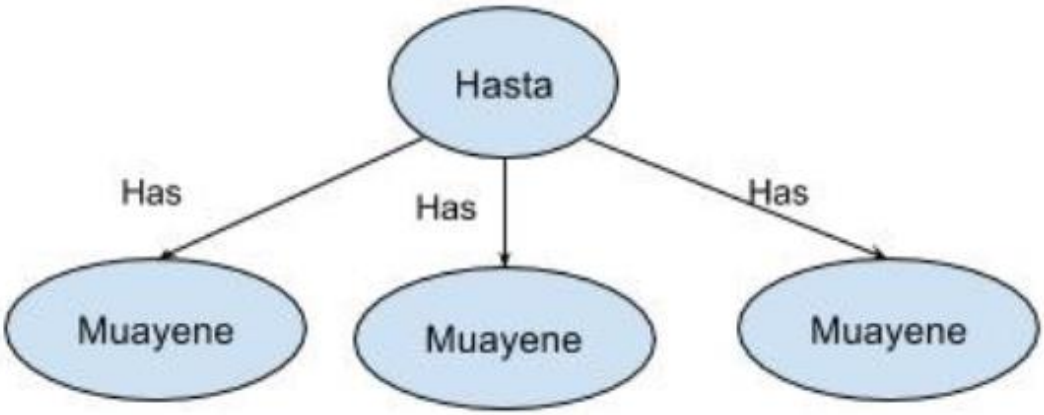
- "KIMLIK_NUMARASI"
- "Ad"
- "Soyad"
- "DOGUM_TARIHI"
- "PROTOKOL_NUMARASI"

Bir Muayene_node vertex için oluşturulan property’ler:

- "MUAYENE_TARIH"
- "DOKTOR_KIMLIK_NUMARASI"
- "TESHIS"

.txt dosyasından alınan veriler .setProperty() (örnek kod: Hasta.setProperty("Ad", ad)) metoduyla vertex’lere atanmış bilgi ataması tamamlanan vertex titanGraph.commit() ile veritabanına kaydedilmiştir.

addEdge("By", doktor_node) ile Muayene_node ve Hasta vertex’leri arasında “has” ilişkisi tanımlanmıştır.



Şekil 2-1: Hasta kayıt veritabanı versiyon 1

Yukarıdaki şekilde gösterildiği gibi “has” ilişkisiyle birbirine bağlı Hasta-Muayene kayıtları üzerinde şu sorgular çalıştırılmıştır:

- **Bir hastanın tüm muayenelerinin listelenmesi:**

```
TitanVertex kisi = (TitanVertex) tx.query().has("KIMLIK_NUMARASI", "7")
.has("Protokol Numarası", "P7")
                    .vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver1 = g.V(kisi).out("Has").toList();
            int c = 0;
            do {
                System.out.println("Muayene tarihi:" + ver1.get(c)
                .value("MUAYENE_TARIH"));
                c++;
            } while (ver1.size() != c);
```

Şekil 2-2: Bir hastanın tüm muayenelerinin listelenmesi

Kod parçası gremlin sorgu dilinde tx, titan çizge için açılmış bir transaction’dır. has(KIMLIK_NUMARASI, tc) ile işaret edilen vertex’den out("Has") edge’iyle çıkan düğümler bir listede tutulmuştur ve "MUAYENE_TARIH" property’leri ekrana basılmıştır.

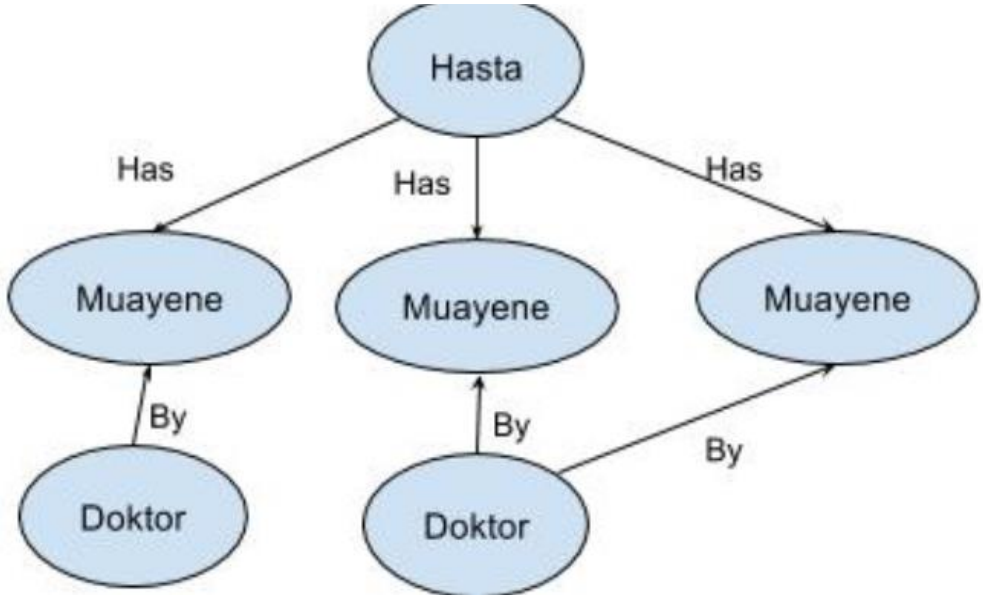
- **Bir tarihte muayeneye gelen tüm hastaların listelenmesi:**

```
Iterable<TitanVertex> ver = tx.query().
has("MUAYENE_TARIH", "21/2/2009").vertices();

            i = 1;
            for (TitanVertex s : ver) {

                List<org.apache.tinkerpop.gremlin.structure.Vertex>
                sonuc = g.V(s).in("Has").toList();
                System.out
                        .println("21/2/2009 tarihinde gelen
                        " + i + ". hasta:" + sonuc.get(0)
                        .value("KIMLIK_NUMARASI"));
                i++;
            }
```

Şekil 2-3: Bir tarihte muayeneye gelen tüm hastaların listelenmesi

has("MUAYENE_TARIH", "21/2/2009") ile "MUAYENE_TARIH" değeri "21/2/2009" olan vertex'ler bir Iterable ile tutlmuştur. Kısıta uyan vertex'leri tutan Iterable'dan sırayla vertex'ler alınmıştır. Bu vertex'lere "Has" ilişkisiyle bağlı olan vertex'ler bir listede tutulmuştur. sonuc adlı listeden teker teker alınan vertex'lerin "KIMLIK_NUMARASI" property'si ekrana basılmıştır.

Bu sorguların ardından çizge veritabanı geliştirilerek şeklideki hali almıştır:



Şekil 2-4: Hasta kayıt veritabanı versiyon 2

Eklenen Doktor vertex için oluşturulan property'ler:

- “DOKTOR_AD”
- “DOKTOR_TC”
- “UZMANLIK”

Bundan sonra devam eden sorgular üç vertex ile ilişkilidir:

- **Bir doktorun tüm muayenelerinin listelenmesi:**

```
TitanVertex dr = (TitanVertex) tx.query().has("DOKTOR_TC", "13")
.vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex> ver2
            = g.V(dr).out("By").toList();
            c = 0;

            do {
                l = 0;
                TitanVertex muayene_v = (TitanVertex) ver2.get(c);
                List<org.apache.tinkerpop.gremlin.structure.Vertex> sonuc2
                = g.V(muayene_v).in("Has").toList();
                do {
                    System.out.println(dr.property("DOKTOR_TC").toString()
                            + " TC kimlikli doktorun muayene ettiği hastalar:"
                            + sonuc2.get(l).value(KIMLIK_NUMARASI));
                    l++;
                } while (sonuc2.size() != l);
                c++;
            } while (ver2.size() != c);
```

Şekil 2-5: Bir doktorun tüm muayenelerinin listelenmesi

dr vertex değişkenine, istenen doktor TC’si aracılığıyla bulunur ve atanır. Doktor vertex’ine “By” ilişkisiyle gelen Muayene vertex’leri bir liste aracılığıyla saklanır. Teker teker her vertex’in bağlı olduğu Hasta vertex’ine ulaşılır ve istenilen bilgiler ekrana yazılır.

- **Bir doktorun belli bir tarihte muayene ettiği hastalar:**

```
TitanVertex dr2 = (TitanVertex) tx.query().has("DOKTOR_TC", "13")
.vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex> ver3
            = g.V(dr2).out("By")
                    .has("MUAYENE_TARIH", "2/5/2009").toList();
            l = 0;
            do {
                c = 0;
                TitanVertex muayene_v = (TitanVertex) ver3.get(l);
                List<org.apache.tinkerpop.gremlin.structure.Vertex> sonuc3
                = g.V(muayene_v).in("Has").toList();
                do {
                    System.out.println(dr.property("DOKTOR_TC").toString()
                        + "13 TC kimlikli doktorun 2/5/2009 tarihinde muayene ettiği hastalar:"
                        + sonuc3.get(c).value(KIMLIK_NUMARASI));
                    c++;
                } while (sonuc3.size() != c);
                l++;
            } while (ver3.size() != l);
```

Şekil 2-6: Bir doktorun belli bir tarihte muayene ettiği hastalar

dr2 vertex değişkenine, istenen doktor TC'si aracılığıyla bulunur ve atanır. Doktor vertex'ine "By" ilişkisiyle gelen ve muayene tarihi belirli olan Muayene vertex'leri bir liste aracılığıyla saklanır. Teker teker her vertex'in bağlı olduğu Hasta vertex'ine ulaşılır ve istenilen bilgiler ekrana yazılır.

- **Belirli TC numaralı hastayı muayene eden doktorlar, tarih sırasına göre sıralı:**

```
TitanVertex kisi2 = (TitanVertex) tx.query().has("KIMLIK_NUMARASI", "7")
.has("Protokol Numarası", "P7")
                    .vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex> ver4
            = g.V(kisi2).out("Has").toList();
            c = 0;
            int sayac = -1;
            int[] sonuclars;
            sonuclars = new int[3];
            do {
                l = 0;
                TitanVertex muayene_ver = (TitanVertex) ver4.get(c);
                List<org.apache.tinkerpop.gremlin.structure.Vertex> sonuc3
                = g.V(muayene_ver).in("By").toList();
                deger = sonuc3.size();
                sayac++;
                TitanVertex doktor_v = (TitanVertex) sonuc3.get(0);
                String string = muayene_ver.property("MUAYENE_TARIH").toString();
                String[] parts = string.split("/");
                String gunler = parts[0];
                String[] gun_int = gunler.split(">");
                int gun = Integer.parseInt(gun_int[1]);
                int ay = Integer.parseInt(parts[1]);
                String yillar = parts[2];
                String[] yil_int = yillar.split("]");
                int yil = Integer.parseInt(yil_int[0]);
                int sonuc = (yil * 10000) + (ay * 100) + gun;
                sonuclars[sayac] = sonuc;
                doktor_list.put(sonuclars[sayac], doktor_v);
                l++;
                c++;
            } while (ver4.size() != c);

            int k;
            int temp;
            for (int m = sayac; m >= 0; m--) {
                for (int ii = 0; ii < sayac; ii++) {
                    k = ii + 1;
                    if (sonuclars[ii] > sonuclars[k]) {

                        temp = sonuclars[ii];
                        sonuclars[ii] = sonuclars[k];
                        sonuclars[k] = temp;
                    }
                }

            }
            c = 0;

            System.out.println("7 TC numaralı hastayı muayene eden doktorlar,
            tarih sırasına göre sıralı");
            while (c != 2) {

                TitanVertex doktor_node = doktor_list.get(sonuclars[c]);
                System.out.print(sonuclars[c]);
                System.out.println(doktor_node.property("DOKTOR_TC").toString());
                c++;
            }
```

Şekil 2-7: Belirli TC numaralı hastayı muayene eden doktorlar, tarih sırasına göre sıralı

TC numarasıyla belirlenen kişi, kisi2 değişkeninde tutulur. Bu kişiden "Has" ilişkisiyle çıkan Muayene vertex'leri bir listede tutlur. Bu günlerin hangi hastaya ait olduğunu anlamak için bir linkedhasmap'e Doktor vertex'i ve o tarih atılır. Muayene vertex'leri gün olarak sıralanmak için bir dizi işlemden geçirilir ve

sonuclars adlı dizi sıralı günleri tutar. Sonuçta listedeki doktorlar, gün sırasına göre teker teker çekilir ve istenen veriler ekrana yazılır.

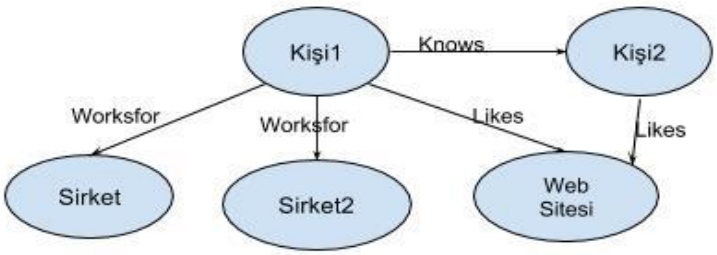
- **Belli bir tarihte muayene yapan doktorlar ve ilgili muayenedeki hasta:**

```
Iterable<TitanVertex> ver5 = tx.query().has("MUAYENE_TARIH", "21/2/2009")
.vertices();

            i = 1;
            for (TitanVertex s : ver5) {

                List<org.apache.tinkerpop.gremlin.structure.Vertex> sonuc
                = g.V(s).in("Has").toList();
                System.out.println(
                        "21/2/2009 tarihinde gelen " + i + ". hasta TC:" +
                        sonuc.get(0).value("KIMLIK_NUMARASI"));
                i++;
                List<org.apache.tinkerpop.gremlin.structure.Vertex> doktor
                = g.V(s).in("By").toList();
                System.out.println("Hastayı muayene eden doktor TC :" +
                doktor.get(0).value("DOKTOR_TC").toString());
            }
```

Şekil 2-8: Belli bir tarihte muayene yapan doktorlar ve ilgili muayenedeki hasta

İstenilen tarihli muayeneye has("MUAYENE_TARIH", "21/2/2009") ile ulaşılır. O tarihli tüm muayeneler bir Iterable'da saklanır. Bu vertex'ler teker teker alınır ve "Has" ilişkisiyle kendilerine gelen Hasta vertex'i bir listede tutulur. Buradan hastanın TC numarasına ulaşıldıktan sonra, Muayene vertex'inden çıkan "Has" ilişkisiyle Doktor vertex'i bir listede tutulur. Buradan da hastayı muayene eden doktora ulaşılarak istenen veriler ekrana yazılır.

2.2. Person Veritabanı



Şekil 2.9: Person veritabanı çizge mantığı

Person veritabanı; kişiler, şirketler ve web sitelerini saklayan ayrıca bunların birbirleri arasındaki etkileşimleri sunan çizge veritabanıdır. Kişiler tanıdıkları kişilere ‘Knows’, çalıştıkları şirketlere ‘Worksfor’ ve beğendikleri web sitelerine ‘Likes’ ilişkileriyle bağlanmıştır. Veritabanında bulunan ‘Kisi’ Vertex’i şu property’leri içermektedir:

- "KIMLIK_NUMARASI"
- "Ad"
- "Soyad"

‘Sirket’ Vertex’i şu property’leri içermektedir:

- "SIRKET_ADI"

‘Site’ Vertex’i şu property’leri içermektedir:

- "SITE_ADI"

Tanımlanan Vertex’lerde saklanan örnek verilerle aşağıdaki sorgular çalıştırılmıştır:

- **İsmi X olan kişinin tanıdığı kişiler:**

```
TitanVertex kisi = (TitanVertex) tx.query().has("Ad", "hasta3")
.vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver1 = g.V(kisi).out("Knows").toList();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver2 = g.V(kisi).in("Knows").toList();
            int c = 0;
            if (ver1.size() != 0)
                do {

                    System.out.println("Adı:" + ver1.get(c).value("Ad"));
                    c++;
                } while (ver1.size() != c);

            c = 0;
            if (ver2.size() != 0)
                do {
                    System.out.println("Adı:" + ver2.get(c).value("Ad"));

                    c++;
                } while (ver2.size() != c);
```

Şekil 2-10: İsmi X olan kişinin tanıdığı kişiler

TitanVertex kisi ile tanımlanan kisi vertex'ine rasgele hasta3 kişisinin bilgileri kaydedilmiştir. Veritabanı kişilere in ve out olmak üzere iki farklı 'Knows' ilişkisi içermektedir. Bu sebeple kişinin tanıdıkları ele alınırken hem out hem in için iki farklı senaryo izlenmiştir. List<org.apache.tinkerpop.gremlin.structure.Vertex> ver1, kişiden diğer kişiye oluşturulmuş 'Knows' ilişkisine sahip vertex'leri; ver2 ise kişiye 'Knows' ile gelen diğer kişi vertex'i tutar. Liste şeklinde tutulan tüm tanıdıklar bir do-while döngüsüyle teker teker alınır. ver1.get(c).value("Ad") değişkeniyle istenilen tanıdık bilgisine ulaşılır.

- **X isimli kişinin 1. Derecede tanıdıkları nerelerde çalışıyor?**

```
TitanVertex kisi2 = (TitanVertex) tx.query().has("Ad", "hasta3")
.vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver3 = g.V(kisi2).out("Knows").toList();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver4 = g.V(kisi2).in("Knows").toList();
            c = 0;
            if (ver3.size() != 0)
                do {

                    System.out.println("Adı:" + ver3.get(c).value("Ad"));
                    tanidik = (TitanVertex) ver3.get(c);
                    List<org.apache.tinkerpop.gremlin.structure.Vertex>
                    sirket_list = g.V(tanidik).out("Worksfor")
                            .toList();
                    int m = 0;
                    if (sirket_list.size() == 0)
                        System.out.println("Çalışmıyor...");
                    do {
                        System.out.println("Çalıştığı şirket:"
                        + sirket_list.get(m).value("SIRKET_ADI"));
                        m++;
                    } while (sirket_list.size() != m);
                    c++;
                } while (ver3.size() != c);

            c = 0;
            if (ver4.size() != 0)
                do {
                    tanidik = (TitanVertex) ver4.get(c);
                    System.out.println("Adı:" + tanidik.value("Ad"));
                    List<org.apache.tinkerpop.gremlin.structure.Vertex>
                    sirket_list = g.V(tanidik).out("Worksfor")
                            .toList();
                    int k = 0;
                    if (sirket_list.size() == 0)
                        System.out.println("Çalışmıyor...");
                    else
                        do {
                            System.out.println("Çalıştığı şirket:"
                            + sirket_list.get(k).value("SIRKET_ADI"));
                            k++;
                        } while (sirket_list.size() != k);
                    c++;
                } while (ver4.size() != c);
```

Şekil 2-11: X isimli kişinin 1. Derecede tanıdıkları nerelerde çalışıyor?

İlk sorgudaki benzer mantık burada da uygulanmıştır. ver3 ve ver4, kişinin tanıdığı Vertex'leri saklayan listelerdir. List<org.apache.tinkerpop.gremlin.structure.Vertex> sirket_list 'Worksfor' ilişkisiyle bağlı olan Şirket Vertex'lerini tutan listedir. Bu liste ile şirketlere erişilir ve eğer çalışıyorsa çalıştığı şirketin adı ekrana yazdırılır.

- **X isimli kişinin 2. Derece tanıdıkları nerelerde çalışıyor?**

```
TitanVertex kisi3 = (TitanVertex) tx.query().has("Ad", "hasta3")
.vertices().iterator().next();
            //ismi x olanların tanıdıgı kisiler
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver5 = g.V(kisi3).out("Knows").toList();
            //ismi x olanları tanıyan kisiler
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver6 = g.V(kisi3).in("Knows").toList();

            if (ver5.size() != 0)
                loops(ver5, kisi3, g);
            if (ver6.size() != 0)
                loops(ver6, kisi3, g);
```

```
public void loops(List<org.apache.tinkerpop.gremlin.structure.Vertex>
 ver5, TitanVertex kisi3,
            GraphTraversalSource g) {
        int c = 0;
        do {
            //x in 2.derece tanıdıkları
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver7 = g.V(ver5.get(c)).out("Knows").toList();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            ver8 = g.V(ver5.get(c)).in("Knows").toList();

            if (ver7.size() > 0)
                loops2(ver7, kisi3, g);

            if (ver8.size() > 0)
                loops2(ver8, kisi3, g);
            c++;
        } while (ver5.size() != c);
    }
```

Şekil 2-12: X isimli kişinin 2. Derece tanıdıkları nerelerde çalışıyor?

```
public void loops2(List<org.apache.tinkerpop.gremlin.structure.Vertex>
ver7, TitanVertex kisi3,GraphTraversalSource g) {
    int n = 0, m = 0;
    do {
        tanidik = (TitanVertex) ver7.get(n);
        if (!tanidik.equals(kisi3)) {
            System.out.println("Adı:" + tanidik.value("Ad"));
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            sirket_list = g.V(tanidik).out("Worksfor").toList();

            if (sirket_list.size() == 0)
                System.out.println("Çalışmıyor...");
            else{
                do {
                    System.out.println("Çalıştığı şirket:"
                    + sirket_list.get(m).value("SIRKET_ADI"));
                    m++;
                } while (sirket_list.size() != m);}

        }
        n++;
    } while (ver7.size() != n);
}
```

Şekil 2-13: X isimli kişinin 2. Derece tanıdıkları nerelerde çalışıyor?

Bir kişinin 2. derece tanıdıkları için öncelikle tanıdıkları ver5 ve ver6 olmak üzere 2 listede tutulmuştur. loops fonksiyonu ile kişinin 2. dereceden tanıdıklarına ulaşılmıştır. Bunun için ver7 ve ver8 listeleri kullanılmıştır. Kişiyi tanıyanları tanıyanların tutulduğu bu listeler loops2 fonksiyonuna gönderilerek, 2. derece tanıdıkların çalıştığı şirketlere ulaşılmıştır.

- **Bir şirkette çalışanların listelenmesi**

```
TitanVertex sirket1= (TitanVertex) tx.query().has("SIRKET_ADI", "Sirket2")
.vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            sir_list = g.V(sirket1).in("Worksfor").toList();
            System.out.println("Sirket 2 de calisanlar:");
            do{

                System.out.println("Kimlik numarasi: "
                +sir_list.get(c).value(KIMLIK_NUMARASI));
                c++;
            }while(sir_list.size()!=c);
```

Şekil 2-14: Bir şirkette çalışanların listelenmesi

Bir şirkette çalışanlar için sirket1 Vertex'inde o şirketin bilgileri tx.query().has("SIRKET_ADI", "Sirket2").vertices().iterator().next() sorgusu ile

getirilmiştir. sir_list, o şirket için çalışan kişilere ‘Worksfor’ ilişkisiyle bağlanan kişi Vertex’lerini tutan listedir. Bu liste aracılığıyla şirkette çalışan kişi bilgilerine ulaşılır.

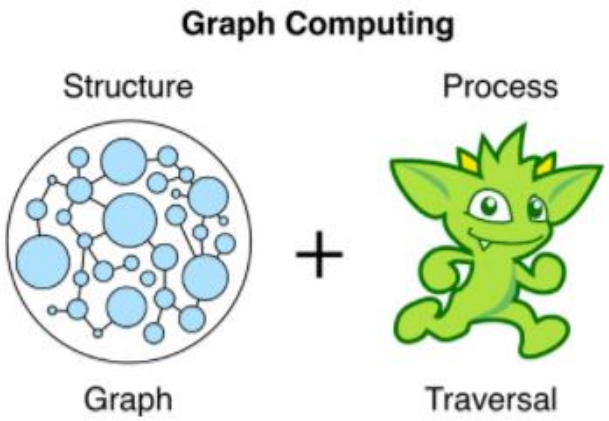
- **X isimli sitesinden hoşlananlar nerede çalışıyor?**

```
TitanVertex site1= (TitanVertex) tx.query().has("SITE_ADI", "MSN")
.vertices().iterator().next();
            List<org.apache.tinkerpop.gremlin.structure.Vertex>
            site_list = g.V(site1).in("Likes").toList();
            c=0;
            do{
                System.out.println("Kimlik numarasi: "
                +site_list.get(c).value(KIMLIK_NUMARASI));
                c++;
            }while(site_list.size()!=c);
```

Şekil 2-15: X isimli sitesinden hoşlananlar nerede çalışıyor?

site1 ile rastgele bir site Vertex’i tutulur. site_list, ‘Likes’ ilişkisiyle bağlanan kişi Vertex’lerini tutan listedir. Bu liste aracılığıyla istenilen kişi bilgilerine ulaşılır.

2.3. Çizge Üzerinden Dolaşım



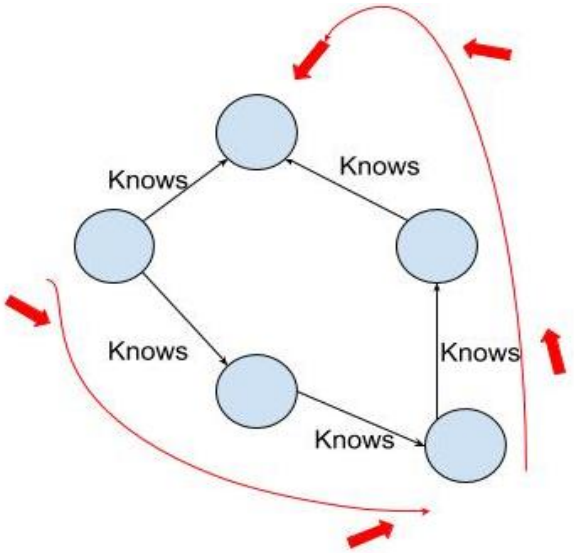
Şekil 2-16: Çizge işleme yapı taşları

Çizge işleme süreci yapısal çizge ve “traversal” yani gezinme sürecinin birleşmesinden oluşur. Çizge düğüm ve ilişkilerden oluşan bir veri modelidir. Çizge işlemenin temel mantığı bu yapının analiz edilmesidir. Çizge işlemenin tipik formu gezinme olarak adlandırılır.

```
TitanVertex kisi4 = (TitanVertex) tx.query().has("Ad", "hasta3")
.vertices().iterator().next();
@SuppressWarnings("unchecked")
List<Vertex> ver01 = g.V().has("Ad", "hasta1").repeat(__.in("Knows"))
.until(__.in("Knows").count().is(0)).toList();
List<Vertex> ver02 = g.V().has("Ad", "hasta1").repeat(__.out("Knows"))
.until(__.out("Knows").count().is(0)).toList();
c=0;
if(ver01.size()!=0)
do {
    TitanVertex temp = (TitanVertex) ver01.get(c);
    System.out.println("Adı: "+temp.value("Ad"));
    List<Vertex> temp2 = g.V(temp).out("Likes").toList();
    int m=0;
    do{
        System.out.println("Begendigi Sirket Adı:" + temp2.get(m)
        .value("SITE_ADI").toString());
        m++;
    }while(temp2.size()!=m);
    c++;
} while (ver01.size() != c);
c=0;
if(ver02.size()!=0)
do {
    TitanVertex temp=(TitanVertex) ver02.get(c);
    System.out.println("Adı: "+temp.value("Ad"));
    int m=0;
    List<Vertex> temp2 = g.V(temp).out("Likes").toList();
    do{
        System.out.println("Begendigi Sirket Adı:" + temp2.get(m)
        .value("SITE_ADI").toString());
        m++;
    }while(temp2.size()!=m);
    c++;
} while (ver01.size() != c);
```

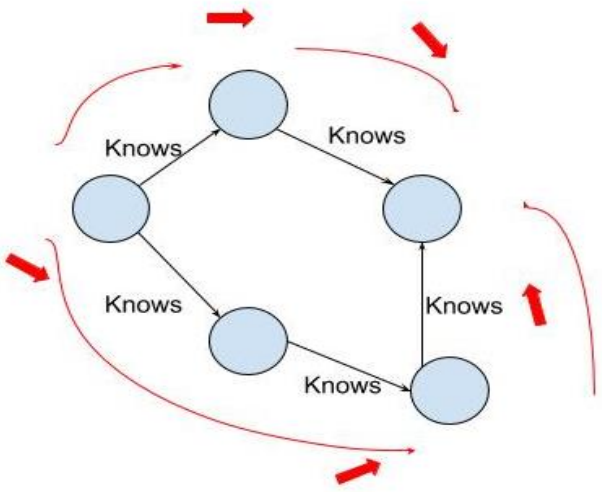
Şekil 2-17: Örnek kod parçası

Yukarıda gösterilen kod parçası, düğümler arasında gezinerek son düğüme kadar gelir.



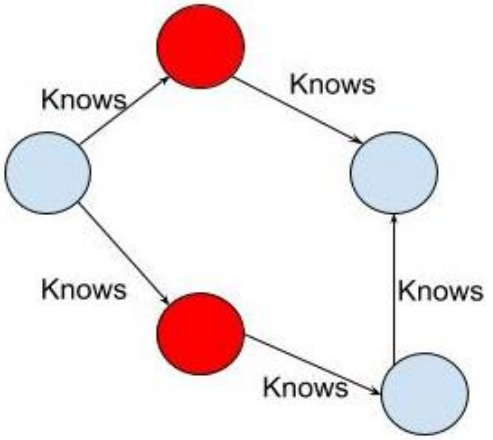
Şekil 2-18: Çizge gezinimi

Kod parçasında anlatılan süreç şekildeki akışı izler. repeat(__.in("Knows")) kod parçasıyla, “Knows” edge’inin düğüme girdiği yönden dolaşmaya başlayacağını belirtir. Düğümlere giren ilişkiler, until(__.in("Knows").count().is(0)) koduyla sonlanır. Buradaki sonlanma koşulu düğüme giren hiçbir “Knows” ilişkisi kalmamış olmasıdır.



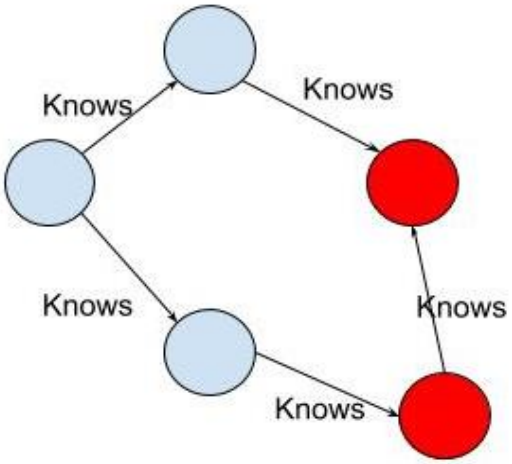
Şekil 2-19: Çizge gezinimi -2

Ayrıca giren ve çıkan ilişkileri ayırmadan, Şekil 4-19’da gösterildiği gibi tümü kapsayan bir dolaşım da Titan DB ile mümkündür. Üstte anlatılan koddan farklı olarak repeat(__.both("Knows")) kod parçası şekilde gösterilen dolaşımı mümkün kılar.



Şekil 2-20: Belirli bir derinliğe inme

Titan DB ile farklı derinliklerde gezinim sağlanabilir. Kişinin n. dereceden tanıdıkları ayrı ayrı ele alınabilir. Üstteki kod parçasında bulunan ver01, List<Vertex> ver01 = g.V().has("Ad", "hasta1").repeat(__.both("Knows")).times(1).toList() kod parçası ile değiştirilerek 1. derece tanıdıkların listesine ulaşılır.



Şekil 2-21: Belirli bir derinlikte gezinim

.repeat(__.both("Knows")).times(2) kod parçası 2. dereceden tanıdıklara erişimi sağlar.

BÖLÜM 3. Geleneksel Veritabanı İle Kıyaslama

3.1. Geleneksel Veritabanı İle Hasta Girdi Sistemi Nosql Sorgularının Kıyaslanması

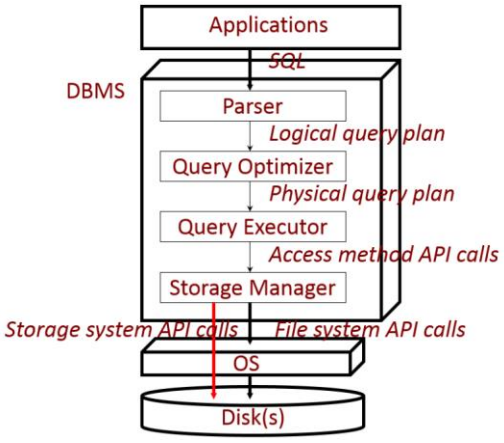
Bundan sonraki adımda verilerin geleneksel bir veritabanında saklanması senaryosu anlatılmıştır.

Geleneksel veritabanları büyük veri için etkin bir yol değildir. Büyük veri için etkin veritabanı yapılarından biri olan çizge veritabanı yapısı geleneksele kıyasla şu temel avantajlara sahiptir:

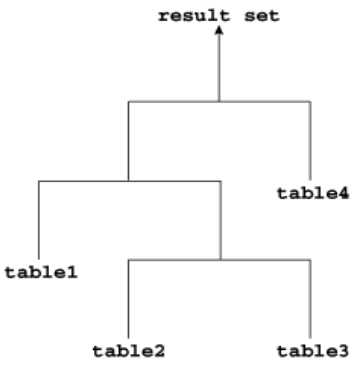
- Şema esnekliği: Mevcut şemalar değiştirilebilir.
- "JOIN" operasyonu fazlalığından kaçınır.
- Daha sezgisel sorgulama: Çoğu NoSQL veritabanı kendi sorgulama diline sahiptir. Bu, sorguların daha sezgisel olmasını sağlar. [2]

Çizge veritabanlarında sorgu süresi çizgeyi oluşturan düğümlerin sayısından etkilenmez. Çünkü çizge veritabanı sorguları, tabloları taramak zorunda değildir, daha fazla düğümün olması çoğu durumda sorgu süresini etkilemez.[2]

Geleneksel veritabanı işleyiş şeması ve Join operasyonu mantığı şöyledir: [3]

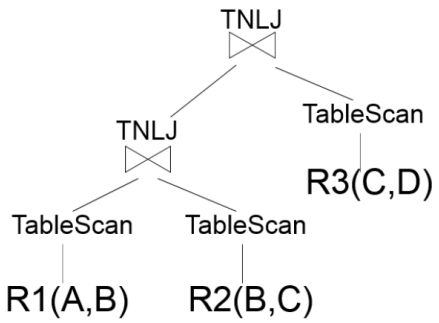


Şekil 3-1:Modern DBMS mimarisi



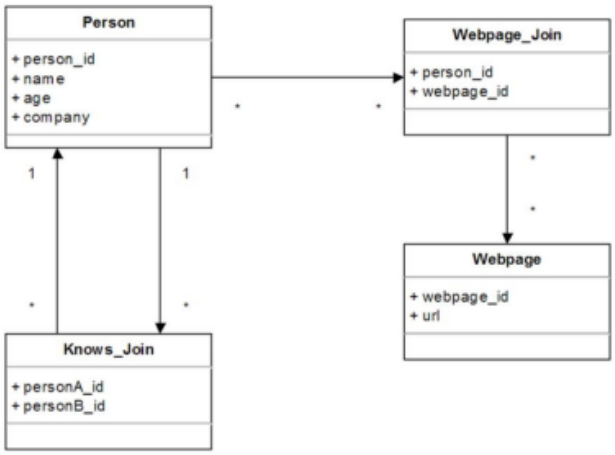
Şekil 3-2: Join Tree

Join operasyonu ağaç yapısı sadece sağdan ya da sadece soldan dallanarak da geliştirilebilir. Sorgu recursive mantıkta işletilir. [3]



Şekil 3-3: Örnek Join Tree

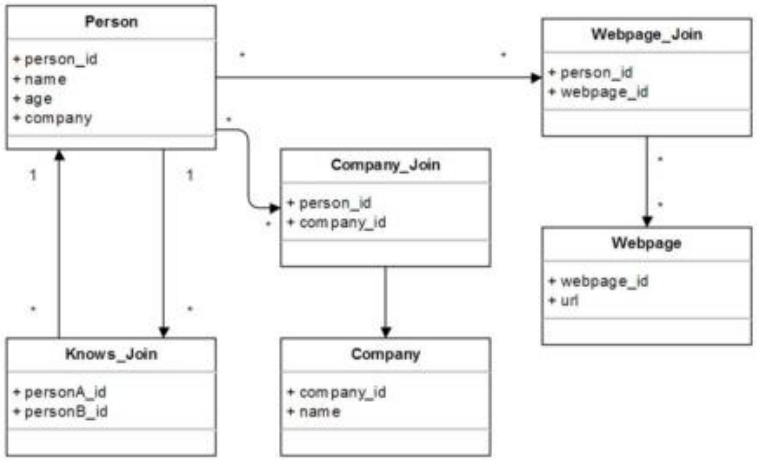
Şekildeki örnekte birbiriyle ilişkili tablolar arasındaki sorgular için gereken tablo taraması gösterilmiştir. A,B,C ve D tabloları arası ilişkinin olduğu bir sorgu, her çocuk düğüm için gerekli olan tablo taraması gerektirir. Bu, sorgunun maliyetini arttırır.[3]



Şekil 3-4: Örnek geleneksel veritabanı

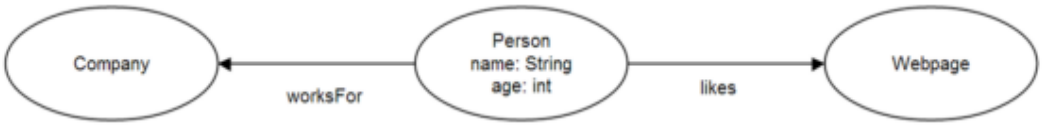
Yukarıda verilen tablo bir şirkette çalışan insanları ve onların web sitelerini saklamaktadır. Knows ve likes ilişkileri join tabloları aracığı ile tutulmaktadır. Bu şema bir kişi için yalnızca bir şirket tutmaktadır. Bu kişiye

ikinci bir şirket ataması yapabilmek için ya tekrarlayan satırlara izin vermek ya da yeni bir join tablosu oluşturmak gerekir. [2]



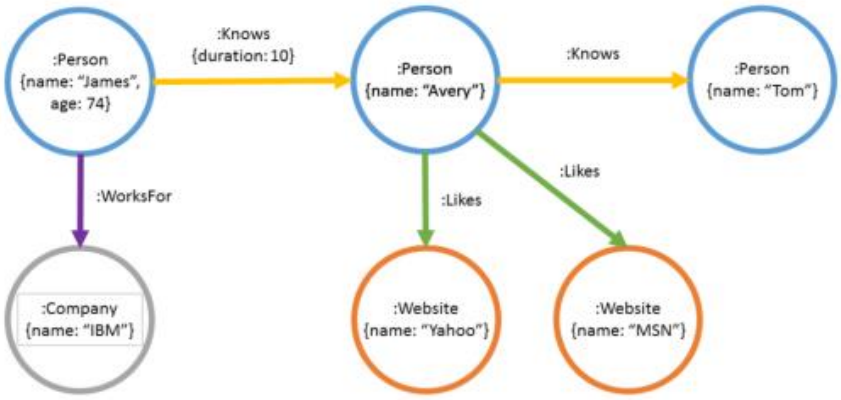
Şekil 3-5: Örnek veritabanı 2. versiyon

Sonuçta eklenen yeni bir join tablosuyla şema, bir kişiye birden çok şirket atayabilir hale geldi. Eklenmek istenen her ilişki, yeni bir join tablosu olarak şemaya eklenmelidir. Bu eklenen join tablolarının sayısı artmaya başladığında problemler oluşmaya başlar. Join tabloları edge'leri temsil eder, birçok join tablosunu barındıran bir veritabanı ile anlamlı bir çalışma yapabilmek için geçişler (gezinme) gerekir. Bu durum sorgu performansının düşmesine neden olur.[2]



Şekil 3-6: Örnek çizge veritabanı

Önceki örnekte temsil edilen yapı bu sefer çizge veritabanıyla saklanmıştır. Aynı problem için yapılması gereken tek şey yeni bir edge'i çizgeye eklemektir. Burada önemli bir nokta şudur; çizge veritabanları, join'leri her edge geçişi için çalıştırmak zorunda değildir. Ayrıca yapılan iyileştirme çalışmalarıyla NoSQL veritabanlarından en üst seviyede verimi almak mümkündür. [5]



Şekil 3-7: Örnek çizge veritabanı 2

Yapılan çalışmalarda 30.000 düğümlü ve 120.000 ilişkili Neo4j çizge veritabanı üzerinde basit sorgular yapılarak süreler hesaplanmıştır. Çizge veritabanı yapısı şekildeki gibidir. [5]

```
START p = node(*)
MATCH (p {name: "James"})-[:Knows]->(friend)
RETURN friend
```

Şekil 3-8: Örnek Cypher sorgusu

Şekilde Cypher sorgu diliyle yazılmış sorgu, tüm düğümleri arayarak o düğümün James'i tanıyıp tanımadığını sorgular. 30.000 düğümlü ve 120.000 ilişkili bu yapıda sorgu süresi 150 ms. olarak hesaplanmıştır. [5]

```
MATCH (p:Person {name: "James"})-[:Knows]->(friend)
RETURN friend
```

Şekil 3-9: Örnek Cypher sorgusu 2

Label eklenerek yapılan sorgular (ilgili düğümlere Person label'ı eklenmiştir) tüm Person düğümlerini arayacak şekilde özelleştirilmiş ve sorgu süresi 80 ms'ye indirilmiştir. [5]

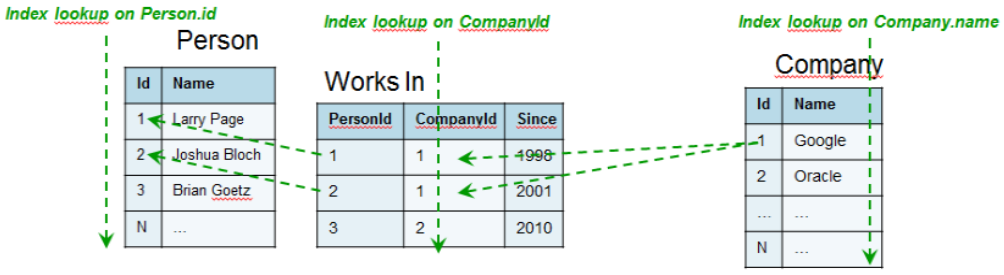
Çizge veritabanları, koşullu bir şekilde değil çizgenin kendisinin sağladığı doğal bitişiklilik index tekniğine sahiptir. Bir düğüme eklenen ilişki aracılıyla ilişkili diğer düğüm ile aralarında doğal bir bağlantı kurulur.[5]

```
MATCH (p:Person {name: "James"})-[:Knows]->(friend)
RETURN friend
```

Şekil 3-10: Örnek Cypher sorgusu 3

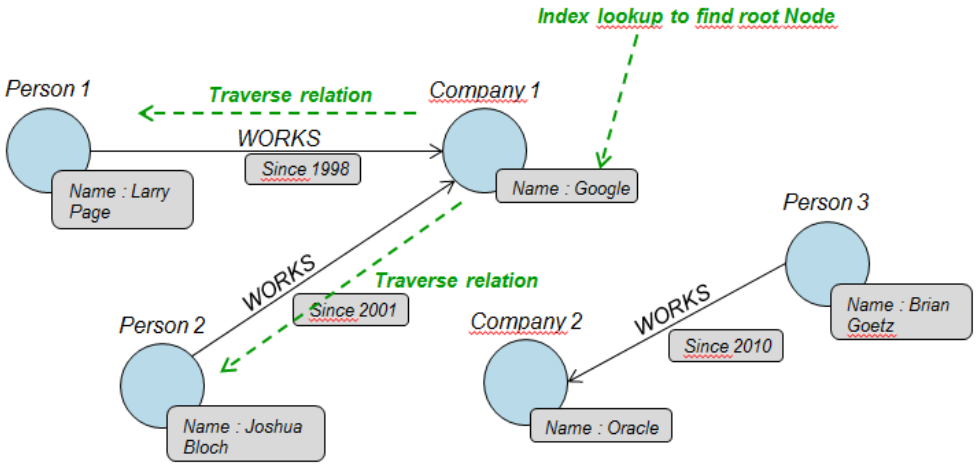
Person(name)'e eklenen index, aramayı lineerden logaritmiğe çevirmiştir. Bu sayede sorgu süresi 6 ms'ye düşmüştür. Index mekanizmasını geleneksel veritabanları ve NoSQL veritabanlarından biri olan çizge veritabanı için kıyaslamak gerekirse: Şirketlerin, şirket çalışanlarının ve onların ne kadar süreyle çalıştıklarının saklanacağı bir yapıyı iki veritabanı için index ile gerçekleştirelim. İstenilen sorgu tüm Google çalışanlarını bulmak olsun. [5]

Geleneksel veritabanı için bu sorgu üç index ile gerçekleştirilebilir.



Şekil 3-11:Örnek index yapısı

Çizge veritabanında gelenekselin aksine 1 index ile bu sorgu gerçekleştirilebilir. [6][7][8]



Şekil 3-12: Çizge veritabanlarında index yapısı

Şekildeki yapı çizge veritabanlarında index işleminin nasıl yapıldığını açıklamaktadır. Ayrıca kullanımları dışında geleneksel veritabanları ve çizge veritabanında index yapısal olarak farklı çalışmaktadır. [6][7][8]

Index konusu çizge veritabanlarından biri olan Titan Veritabanı için ele alınacak olursa; Titan veritabanı iki çeşit index türünü destekler: Graph index ve vertex-centric index. Çoğu sorgu tüm vertex'lerin dolaşılmasını (çizge dolaşımı) içerir. Graph index'ler, global index yapılarıdır ve büyük veritabanlarında gerçekleşecek bu sorguların daha verimli olmasını sağlar. [6][7][8]

Q1:g.V.has('name','hercules')
g.E.has('reason',CONTAINS,'loves')

Yukarıda verilen örnekte ilk sorgu "hercules" isimli tüm vertex'leri ister. Vertex ve edge'ler üzerindeki property'ler anahtar-değer çiftleridir. Mesela name="hercules" property'si için anahtar "name" ve değer "hercules"'dir. Bu, property key olarak adlandırılır. İkinci sorgu "reason" property'si "loves" kelimesini içeren tüm edge'leri sorgular. Graph index olmadan bu sorguları gerçekleştirmek için tüm vertex ve edge'lerin taranması gerekir ve bu efektif bir yöntem değildir. [6][7][8]

Titan veritabanında graph index ikiye ayrılır: Composite index ve mixed index. Composite index'ler çok hızlı ve efektiftir ancak özel, önceden tanımlı property key kombinasyonları için limitlidir. Sadece equality constraint ("has") içeren sorguları destekler. [6][7][8]

```
mgmt = g.getManagementSystem()
name = mgmt.makePropertyKey('name').dataType(String.class).make()
age = mgmt.makePropertyKey('age').dataType(Integer.class).make()
mgmt.buildIndex('byName',Vertex.class).addKey(name).buildCompositeIndex()
mgmt.buildIndex('byNameAndAge',Vertex.class).addKey(name).addKey(age).buildCompositeIndex()
mgmt.commit()
```

Yukarıdaki kod parçasında öncelikle name ve age property key'leri tanımlanımş sonra basit bir composite index, sadece name property key'i üzerine kurulmuştur. Titan Q2'deki sorguyu gerçekleştirmek için bu index'i kullanacaktır.

Q2: g.V.has('name','hercules')

İkinci composite graph index, iki key'i de içermektedir. Titan Q3'deki sorguyu gerçekleştirmek için bu index'i kullanacaktır.

Q3: g.V.has('age',30).has('name','hercules')

Q4: g.V.has('age',30)

Q4, name property key'ini içermediği için Index'ler ile sorgulanamaz.

Q5: g.V.has('name','hercules').interval('age',20,50)

Q5, name üzerine kurulu Simple Composite Index ile cevaplandırılabilir. Bu sorguda age bir equality constraint olmadığı içinComposite Graph Index ile cevaplanamaz. [6][7][8]

Mixed index'ler, herhangi bir property key kombinasyonu için kullanılabilir. Composite index'lere göre daha esnektir ancak çoğu sorguda Composite index'lere göre daha yavaştır. Mixed index, index backend yapılandırmasına ihtiyaç duyar. Bu yapılandırma hangi arama özelliklerinin kullanılacağını tanımlar. Titan Elasticsearch ve Lucene olmak üzere iki index backend'i destekler. Elasticsearch, bulut platformu için açık kaynak, esnek, güçlü ve dağıtık bir gerçek zamanlı arama ve analiz motorudur. Küçük bir boyutta başlayıp büyümeye izin verir. Desteklediği özellikler şunlardır:

Full Text: Bir kelimeyle, önekle veya belirli bir ifadeyle eşleşen kelime property'lerini aramak için tüm metin koşullarını destekler.

Geo: Verilen çember içine düşen bir noktanın konumunun aranmasını destekler.

Numeric: Tüm sayısal karşılaştırmaları destekler.

Lucene, tamamen Java dilinde yazılmış yüksek performanslı metin arama motorudur. Tam metin aramaları için uygundur ve neredeyse tüm teknolojilerle birlikte kullanılabilir. [6][7][8]

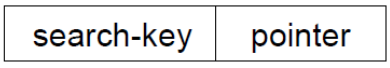
```
mgmt = g.getManagementSystem()
name = mgmt.makePropertyKey('name').dataType(String.class).make()
age = mgmt.makePropertyKey('age').dataType(Integer.class).make()
mgmt.buildIndex('nameAndAge',Vertex.class).addKey(name).addKey(age).build
MixedIndex("search")
mgmt.commit()
```

Yukarıda verilen kod parçası, name ve age property key'leri içeren bir Mixed Index tanımlar. buildMixedIndex("search") kısmı sayesinde Titan hangi index backend'i kullanacağından haberdar olur.

Q6,7,8:
```
g.V.has('name',CONTAINS,'hercules').interval('age',20,50)
g.V.has('name',CONTAINS,'hercules')
g.V.has('age',LESS_THAN,50)
```

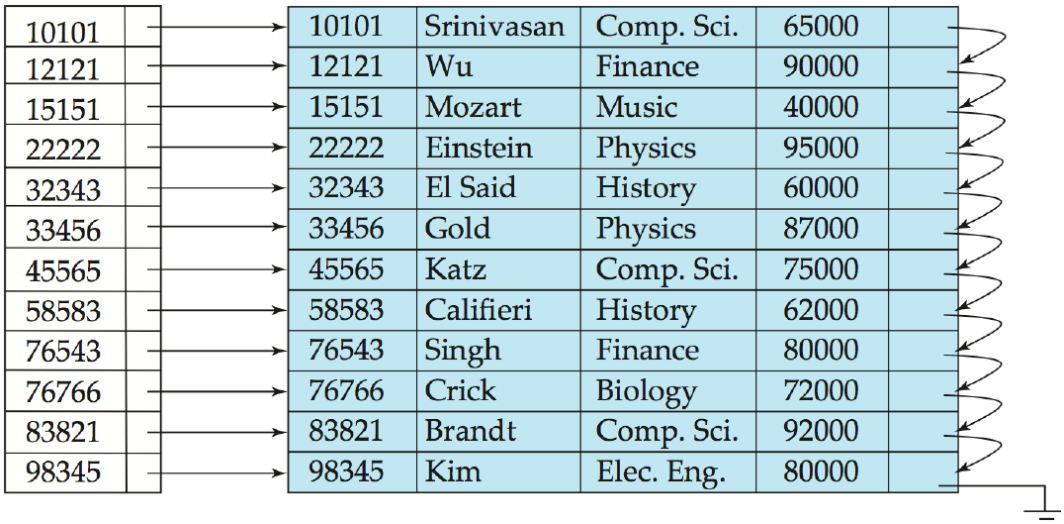
Yukarıda verilen sorgular, Composite Graph Index'in aksine Mixed Index ile desteklenir ve cevaplandırılabilir.[6][7][8]

Geleneksel veritabanlarında index yapısı ise şöyledir: Geleneksel veritabanlarında index yapısı aynı çizge veritabanlarında olduğu gibi sorguları hızlandırmak için kullanılır. [9][10]



Şekil 3-13: Index yapısı

Şekildeki search key,bir dosyada bulunan attribute setinde aranacak olan attribute'ü göstermektedir. Çok çeşitli Indexing tekniği bulunmaktadır. En çok kullanılan tekniklerden biri olan Dense Index'de index kaydı dosyadaki her search-key değeri için gösterilir. [9][10]



Şekil 3-14: Dese Index örneği

Şekilde gösterilen örnekte instructor relation'ının ID attribute'ü üzerinde bir index vardır.[9][10]

Projemiz için bu kıyaslama yapılacak olursa eğer kayıtlar geleneksel bir veritabanında tutulsaydı durum şöyle olacaktı;

KIMLIK_NUMARASI	AD	SOYAD	DOGUM_TARIHI	EMEKLILIK_KODU	PROTOKOL_NUMARASI
1	hasta1	soyad1	1990	1WQOERK2	P1
2	hasta2	soyad2	1941	1WQORFTK2	P2

Çizelge 3-1: Hasta Tablosu

MUAYENE_TARIH	HASTA_TC_NO	DOKTOR_KIMLIK_NUMARASI	TESHIS
5/1/2013	1	12	HastalikB
2/7/2009	2	13	HastalikC

Çizelge 3-2: Muayene Tablosu

DOKTOR_AD	DOKTOR_TC	UZMANLIK
D2	12	Alan2
D3	13	Alan3

Çizelge 3-3: Doktor Tablosu

Verilen tablolar, projede kullanılan veritabanı yapısına uygun olarak tasarlanmıştır. Örnek kayıtlar eklenmiştir. KIMLIK_NUMARASI ve HASTA_TC_NO, DOKTOR_KIMLIK_NUMARASI ve DOKTOR_TC foreign key'lerdir. Bu veriler doğrultusunda yazılacak sorgular şu şekildedir:

- **Bir hastanın tüm muayenelerinin listelenmesi:**

SELECT KIMLIK_NUMARASI,MUAYENE_TARIH

FROM (Hasta **JOIN** Muayene **ON** KIMLIK_NUMARASI=HASTA_TC_NO)

WHERE KIMLIK_NUMARASI='1'

- **Bir tarihte muayeneye gelen tüm hastaların listelenmesi:**

SELECT KIMLIK_NUMARASI, MUAYENE_TARIH

FROM (Hasta **JOIN** Muayene **ON** KIMLIK_NUMARASI=HASTA_TC_NO)

WHERE MUAYENE_TARIH= '5/1/2013'

- **Bir doktorun tüm muayenelerinin listelenmesi**

SELECT DOKTOR_KIMLIK_NUMARASI, MUAYENE_TARIH

FROM (Muayene **JOIN** Doktor **ON** DOKTOR_KIMLIK_NUMARASI=DOKTOR_TC)

WHERE DOKTOR_TC= '13'

- **Bir doktorun belli bir tarihte muayene ettiği hastalar**

SELECT HASTA_TC_NO

FROM (Muayene **JOIN** Doktor **ON** DOKTOR_KIMLIK_NUMARASI=DOKTOR_TC)

WHERE DOKTOR_TC= '13' **AND** MUAYENE_TARIH="2/7/2009"

- **Belirli TC numaralı hastayı muayene eden doktorlar, tarih sırasına göre sıralı**

SELECT DOKTOR_TC_NO

FROM ((Muayene **JOIN** Doktor **ON** DOKTOR_KIMLIK_NUMARASI=DOKTOR_TC)

JOIN Hasta **ON** KIMLIK_NUMARASI=HASTA_TC_NO)

WHERE HASTA_TC_NO= ‘1’

ORDER BY MUAYENE_TARIH

- **Belli bir tarihte muayene yapan doktorlar ve ilgili muayenedeki hasta**

SELECT HASTA_TC_NO, DOKTOR_TC

FROM ((Muayene **JOIN** Doktor **ON** DOKTOR_KIMLIK_NUMARASI=DOKTOR_TC)

JOIN Hasta **ON** KIMLIK_NUMARASI=HASTA_TC_NO)

WHERE MUAYENE_TARIH=”2/7/2009”

Başlangıçta anlatılan geleneksel veritabanlarındaki Join Tree mantığı, yukarıda yazılan sorguların tablo taraması ile gerçekleşmesini gerektirir. Bu NoSQL veritabanı kullanımına kıyasla sorgu performansının düşmesine neden olacaktır.

3.2. Geleneksel Veritabanı ile NoSQL Özyinelemeli Dolaşımın Kıyaslaması

2.3. bölümde anlatılan bir kişinin n. Dereceden tanıdıklarının bulunması, geleneksel bir veritabanı ile gerçekleştirilmek istendiğinde şu tablolarla oluşturulmuş bir veritabanına ihtiyaç duyulur:

Kimlik_Numarasi	Ad	Soyad
1	hasta1	soyad1
2	hasta2	soyad2
3	hasta3	soyad3

Çizelge 3-4: kisi tablosu

kimlik_no	tanidik_tc
1	2
1	3
2	1
3	1

Çizelge 3-5: tanidik tablosu

kimlik_no	calistigi_sirket
1	sirket1
1	sirket2
2	sirket3
3	sirket1

Çizelge 3-6: sirket tablosu

Bir önceki bölümde anlatılan recursive NoSQL sorgusu ilişkisel veritabanıyla yapılsaydı şu sorguyla gerekecekti:

```
SELECT kimlik_no
FROM kisi
INNER JOIN (
        SELECT tanidik_tc
        FROM tanidik
        GROUP BY kimlik_no
        HAVING kimlik_no= ‘1’)
```

2. derece tanıdıkların listelendiği bu sorgu n. dereceden tanıdıklar için istenirse n-1 JOIN’li bir yapı oluşacaktır.

BÖLÜM 4. SONUÇ

Birçok farklı veritabanı içinden ihtiyaca uygun veritabanı türünü seçmek, problemi çözümün ilk adımlarından biridir. Veri setine uygun, kullanılacak diğer teknolojilerle birlikte sorunsuz çalışacak, sorgu performansı güçlü, işlenecek veri türünü başarıyla yönetebilecek bir veritabanı türüyle sağlam, yönetilebilir, bakıma uygun, performansı dengeli bir sisteme ulaşılır.

Sosyal ağlar gibi çoğunlukla aynı türde veriden oluşan veri setlerinde gezinimin efektif sağlanabilmesi gerekir. Geleneksel veritabanı sistemlerinin baştan tanımlı tablolardan oluşan yapısı, JOIN'lerle bağlanan sorgu sistemi, büyük miktarda gelen veriyi işleyememesi bu tür uygulamalar için yetersiz kalmasına neden olur. NoSQL veritabanları büyük miktarda veriyi efektif olarak işlemek için yaratılan veritabanlarıdır. Şemadan bağımsız veri modeliyle esneklik sunar.

Çizge veritabanı, verileri çizge benzeri bir yapıda saklan NoQSL veritabanı türüdür. Çizge yapısındaki düğümlerde istenen veriler saklanırken düğümler arası ilişkiler, bu çizgeden çıkan dallarda tutulur. Ağaç yapısına benzerliği sayesinde yukarıda anlatılan türde veri setleri için uygundur.

Titan veritabanı bir çizge veritabanı teknolojisidir. Kendi türündeki teknolojilerinden farklı olarak index mekanizması bulunur ve dağıtık makinelerde işleme imkân verir. Index, istenilen düğüme daha hızlı ulaşım sağlar. Bu da sorgu süresini kısaltır. Lokal depolama yöntemleri kullanıldığında büyük verinin saklanması sorunu ortaya çıkar. Sürekli artan veri hacmiyle baş edebilecek dağıtık işleme imkân veren sistemiyle Titan DB, bu sorunu ortadan kaldırmıştır.

Titan DB'nin piyasaya yeni giren bir veritabanı teknoloji olması dolayısıyla geliştirme yapan kişi sayısı azdır. Ancak büyük veri yönetim, işleme kapasitesi ve etkinliği diğer çizge veritabanı sistemlerinden daha gelişmiştir. Bu sebeple Titan DB üzerinde yetkinlik kazanmak bu projenin başlıca amacını oluşturmuştur.

Yapılan çalışmalar için lokal bir veritabanı gibi kullanılan Cassandra ile Titan DB bağlanmış ve Titan DB, lokal sunucu modunda kullanılmıştır. Java programlama diliyle iki farklı veritabanı geliştirilmiştir: Hasta Girdi Sistemi isimli veritabanı üç farklı türde (Hasta, Muayene, Doktor) düğüm ve birbiri arası ilişkileri içeren ilk örnektir. Tüm düğümler üzerinde hızlı gezinim ve geleneksel veritabanı ile karşılaştırma örnek sorgular ile açıklanmıştır. Person Veritabanı Kişi, Şirket ve Web Sitesi düğümlerinden oluşan ikinci veritabanı örneğidir. Bu sistemde daha karmaşık sorgular ile gezinim üzerinde durulmuştur. Ayrıca, n. derecedeki düğüme erişim sağlanması burada örneklenmiştir. Geleneksel veritabanı ile gerçekleştirmek için (n-1) JOIN'e ihtiyaç duyulan bu hantal yapı Titan DB ile aşılmıştır.

Bu projede İki farklı veritabanı örneği ile gerçekleştirilen sorgu denemeleri sayesinde Titan DB'nin esnek gezinim mekanizması kanıtlanmıştır. Ayrıca, SQL dilinde yapılan karşılaştırılmalar ile bu dile olan artıları ortaya çıkarılmıştır. Titan DB'nin yönetimi ve sorgu dilinde hedeflenen etkinlik sağlanmıştır.

Projemizde Titan Veritabanı ve Neo4j DB kullanılarak farkları, birbirlerine olan üstünlükleri analiz edilmiştir. Bu kıyaslama yapılırken veritabanlarının yapısal özellikleri ve öğrenme-kullanılabilirlik seviyeleri üzerinde odaklanılmıştır. Neo4j fazla miktardaki sürümlerindeki fonksiyon değişiklikleri yüzünden eski versiyonlarda birbirleriyle uyumlu olmayan birçok özellik bulunmaktadır. Piyasada bulunan örneklerin çoğu, çalıştığımız versiyondan farklı versiyonlarla yazılmış olup büyük yapısal farklar içermektedir. Uygulama geliştirirken bir önceki versiyonun kabul ettiği bir değişken veya fonksiyonu bir diğer sürümdeki fonksiyon reddedebilir. Neo4j ile çalışma yaparken yüz yüze geldiğimiz en büyük sorun bu olmuştur. Titan DB bu konuda daha tutarlı bir yapıya sahiptir. Neo4j`e nazaran daha az versiyon ve sınıf farkı sayesinde piyasada bulunan örneklerden öğrenme sureci daha başarılı olmuştur. Bir diğer yapısal kıyaslama kullandıkları sorgulama dilleri için yapılmıştır. Neo4j Cypher sorgulama dilini kullanmaktadır. Titan DB`nin sorgulama dili Gremlin`dir. Gremlin, Cypher` a göre Java programlama diline entegre edilmesi daha kolay bir yapıya sahiptir. Fonksiyon çağırımları ile istenilen sorgu oluşturulabilir. Aşama aşama isleyen bir süreç ile sorgu gerçekleştirilir. Bu entegrasyon kolaylığı beraberinde karmaşıklığı da getirmektedir. Cypher, SQL benzeri bir mantıkla sorgu yazılmasını mümkün kılar. SQL`deki SELECT FROM WHERE mantığı neredeyse birebir kopyalanmıştır. Sorguyu text halinde çalıştırması, Gremlin`e göre zayıf kalan bir özelliğidir. Sorgu aşamalı olarak ilerlemediği için yazılan sorgunun hatalı kısmını anlamak daha fazla zaman gerektirir. Benchmarking çalışmaları göstermiştir ki Neo4j ve Titan DB, çok büyük olmayan veri setleri için neredeyse ayni surelerde sonuçlar vermektedir. Titan DB, ölçeklenebilirdir yani birçok makinadan toplanan veriyi analiz edebilir. Bizim çalışmamızda iki DB`nin kıyaslanabilmesi için veriler tek bir makinada saklanmalı ve 1000 node ve altı veri setinden oluşmalıdır. SQL` e yatkın bir kişi için bu dilde geliştirme yapmak daha pratik ve hızlı öğrenme sağlayacağından Neo4j uygun secim olacaktır. Piyasadaki örneklerle uyumlu oluşu ve daha başarılı hata mekanizmasıyla Titan DB, Neo4j`nin önüne geçmiştir.

KAYNAKLAR DİZİNİ

[1] **Titan Documentation**, "Chapter 15. Cassandra", http://s3.thinkaurelius.com/docs/titan/0.5.4/cassandra.html (Erişim Tarihi: 10 Ağustos 2016)

[2] **Prad Nelluru, Brahat Naik, Evan Liu**, Bon Koo. "Graph Databases", http://www.cs.utexas.edu/~dsb/cs386d/Projects14/GraphDBPN.pdf (Erişim Tarihi: 7 Aralık 2016)

[3]**ORACLE,** "Database SQL Tuning Guide", https://docs.oracle.com/database/121/TGSQL/tgsql_join.htm#TGSQL94679 (Erişim Tarihi: 21 Nisan 2017)

[4]**Shivnath Babu,** "Advanced Database Systems", https://www.cs.duke.edu/courses/fall08/cps216/Lectures/03_iterators_and_rewrites.ppt (Erişim Tarihi: 23 Şubat 2017)

[5] **Steve Ataky Tsham Mpinda , Lucas Cesar Ferreira , Marcela Xavier Ribeiro , Marilde Terezinha Prado Santos, Department of Computer Science – Federal University of São Carlos,** "EVALUATION OF GRAPH DATABASES PERFORMANCE THROUGH INDEXING TECHNIQUES", http://aircconline.com/ijaia/V6N5/6515ijaia06.pdf (Erişim Tarihi: 25 Şubat 2017)

[6] **Titan Documentation,** "Chapter 5. Schema and Data Modeling", http://s3.thinkaurelius.com/docs/titan/0.5.0/schema.html#_defining_property_keys (Erişim Tarihi: 3 Eylül 2016)

[7] **Titan Documentation,** "Chapter 9. Indexing for better performance", http://s3.thinkaurelius.com/docs/titan/0.5.0/indexes.html (Erişim Tarihi: 27 Ocak 2017)

[8] **Titan Documentation,** "Chapter 21. Elastic Search", http://s3.thinkaurelius.com/docs/titan/0.5.0/elasticsearch.html (12 Mayıs 2017)

[9] **Michel Domenjoud, Thomas Vial, Archi&Techno,** "Graph databases: an overview", http://blog.octo.com/en/graph-databases-an-overview/ (Erişim Tarihi: 3 Mart 2017)

[10] **Database System Concepts, 5th.,** "Chapter 12: Indexing and Hashing" Ed.https://www.cse.iitb.ac.in/~sudarsha/db-book/slide-dir/ch12.pdf (Erişim Tarihi: 29 Nisan 2017)

[11] **IBM,** "Extracting business value from the 4 V's of big data", http://www.ibmbigdatahub.com/sites/default/files/infographic_file/4Vs_Infographic_final.pdf (Erişim Tarihi: 2 Mayıs 2017)

[12]**Petroleum Review | January,** "The 7 pillars of Big Data", 2015https://www.landmark.solutions/portals/0/lmsdocs/whitepapers/the_7_pillars_of_big_data_whitepaper.pdf (Erişim Tarihi: 16 Eylül 2016)

KAYNAKLAR DİZİNİ (devam)

[13] **D. P. Acharjya, Kauser Ahmed P., (IJACSA) International Journal of Advanced Computer Science and Applications, 2016,** "A Survey on Big Data Analytics: Challenges, Open Research Issues and Tools", https://thesai.org/Downloads/Volume7No2/Paper_67-A_Survey_on_Big_Data_Analytics_Challenges.pdf (Erişim Tarihi: 15 Mart 2017)

[14] **CiteSeerX, Penn State-A Public Research University, "**Advanced Search", http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.3372&rep=rep1&type=pdf (Erişim Tarihi: 20 Mart 2017)

[15]**Johannes Zollmann, Rheinische Friedrich Wilhelms Universität,**"NoSQL Databases", bonnhttp://fingon01.iai.uni-bonn.de/_media/teaching/labs/xp/2012b/seminar/10-nosql.pdf (Erişim Tarihi 18 Şubat 2017)

[16] **Michael Hunger, Ryan Boyd & William.,** "The Definitive Guide to Graph Databases for the RDBMS Developer", Lyonhttp://info.neotechnology.com/rs/773-GON-065/images/Definitive-Guide-Graph-Databases-for-RDBMS-Developer.pdf (Erişim Tarihi: 2 Ocak 2017)

[17] **David W. Bates,Suchi Saria, Lucila Ohno-Machado, Anand Shah, Gabriel Escobar.,** "Big Data In Health Care: Using Analytics To Identify And Manage High-Risk And High-Cost Patients", http://content.healthaffairs.org/content/33/7/1123.abstract (Erişim Tarihi: 23 Eylül 2016)

[18] **Kokciyan N, Turkay R, Uskudarli S, Yolum P, Bakir B, Acar B., 2014, "**Semantic description of liver CT images: an ontological approach." https://www.ncbi.nlm.nih.gov/pubmed/25014939 (Erişim Tarihi: 2 Ekim 2016)

[19] **Ivan Merelli,1 Horacio Pérez-Sánchez,2 Sandra Gesing,3 and Daniele D'Agostino., 2014,** "Managing, Analysing, and Integrating Big Data in Medical Bioinformatics: Open Problems and Future Perspectives", https://www.hindawi.com/journals/bmri/2014/134023/ (Erişim Tarihi: 4 Ekim 2016)

[20] **D. P. Acharjya, Kauser Ahmed P.,** "A Survey on Big Data Analytics: Challenges, Open Research Issues and Tools", http://thesai.org/Publications/ViewPaper?Volume=7&Issue=2&Code=IJACSA&SerialNo=67 (Erişim Tarihi: 10 Ekim 2016)

[21] **Teresa Garcia-Valverde, Andrés Muñoz, Francisco Arcas, Andrés Bueno-Crespo, Alberto Caballero., 2014,** "Heart Health Risk Assessment System: A Nonintrusive Proposal Using Ontologies and Expert Rules", https://www.hindawi.com/journals/bmri/2014/959645/ (Erişim Tarihi: 17 Ekim 2016)

[22] **Travis B. Murdoch, Allan S. Detsky., 3 Nisan 2013,** "The Inevitable Application of Big Data to Health Care", http://jamanetwork.com/journals/jama/article-abstract/1674245 (Erişim Tarihi: 2 Kasım 2016)

KAYNAKLAR DİZİNİ (devam)

[23] **IBM. Jimeng Sun, Chandan K. Reddy., 2013,** "Big Data Analytics for Healthcare", https://www.siam.org/meetings/sdm13/sun.pdf (Erişim Tarihi: 15 Kasım 2016)
[24] **Ian Robinson, Jim Webber, Emil Eifrem.,** "Graph Databases"

[25]**Judith Hurwitz, Marcia Kaufman, Fern Halper.,** "Big Data For Dummies"

[26]**Prof. Lizy Kurian John.,** "Performance Evaluation: Techniques, Tools and Benchmarks", https://lca.ece.utexas.edu/pubs/john_perfeval.pdf

[27]**Sotirios Beis, Symeon Papadopoulos, Yiannis Kompatsiaris.,** "Benchmarking graph databases on the problem of community detection", https://pdfs.semanticscholar.org/73dd/7060a97f8ae5728ac2533926aee492400261.pdf

[28]**Marek Ciglan, Alex Averbuch, Ladialav Hluchy.,** "Benchmarking traversal operations over graph databases", http://ups.savba.sk/~marek/papers/gdm12-ciglan.pdf

ÖZGEÇMİŞ

Özge ERTEN 25.03.1993 yılında Konak – İzmir'de doğmuştur. İlk ve ortaöğretimini Hâkimiyet-i Milliye İlköğretim Okulu'nda tamamlamıştır. 2007 yılında Övgü Terzibaşıoğlu Anadolu Lisesi'ne girmiştir. Lise hayatı boyunca Bilgisayar Mühendisi olmayı hedeflemiş ve bu hedef için çalışmalarını sürdürmüştür. 2011 yılında Ege Üniversitesi Bilgisayar Mühendisliği Bölümü'nü kazanmıştır. Lisans eğitiminde veritabanı, programlama dilleri ve bilgisayar ağları konularına yoğunlaşmıştır. 2015 yılında yine aynı bölümde Yüksek Lisans eğitimine başlamıştır.

Alcatel – Lucent Teletaş ve NATO Allied Land Command firmalarında çalışmıştır.

Keman ve gitar çalmak boş zamanlarını doldurduğu hobileri arasındadır. Ayrıca, taşların üzerine minyatür resimler yapmakla ilgilenmektedir.