EGE ÜNİVERSİTESİ

EÜ FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

NOSQL VERİTABANLARI İÇİN ORTAK SORGU İŞLETİM KATMANININ GELİŞTİRİMİ

Bahtiyar ERDEN

Tez Danışmanı : Prof. Dr. Oğuz DİKENELLİ

Bilgisayar Mühendisliği Anabilim Dalı

Sunuş Tarihi : 08.08.2018

Bornova-İZMİR
2018

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

(YÜKSEK LİSANS TEZİ)

NOSQL VERİTABANLARI İÇİN ORTAK SORGU İŞLETİM KATMANININ GELİŞTİRİMİ

Bahtiyar ERDEN

Tez Danışmanı: Prof. Dr. Oğuz DİKENELLİ

Bilgisayar Mühendisliği Anabilim Dalı

Sunuş Tarihi: 08.08.2018

Bornova-İZMİR
2018

Bahtiyar ERDEN tarafından **YÜKSEK LİSANS TEZİ** olarak sunulan "**NOSQL VERİTABANLARI İÇİN ORTAK SORGU İŞLETİM KATMANININ GELİŞTİRİMİ**" 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 **08.08.2018** tarihinde yapılan tez savunma sınavında aday oybirliği/oyçokluğu ile başarılı bulunmuştur.

Jüri Üyeleri:	**İmza**
Jüri Başkanı: Prof. Dr. Oğuz DİKENELLİ	.....
Raportör Üye: Doç. Dr. Rıza Cenk ERDUR	.....
Üye: Dr. Öğr. Üyesi Selma TEKİR	.....

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

ETİK KURALLARA UYGUNLUK BEYANI

EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliğinin ilgili hükümleri uyarınca Yüksek Lisans Tezi olarak sunduğum "**NOSQL VERİTABANLARI İÇİN ORTAK SORGU İŞLETİM KATMANININ GELİŞTİRİMİ**" 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 her türlü yasal sonucu kabul edeceğimi beyan ederim.

09/08/2018

İmzası

Adı-Soyadı

Fahriye Erden

ÖZET

NOSQL VERİTABANLARI İÇİN ORTAK SORGU İŞLETİM KATMANININ GELİŞTİRİMİ

ERDEN, Bahtiyar

Yüksek Lisans Tezi, Bilgisayar Mühendisliği Anabilim Dalı
Tez Danışmanı: Prof. Dr. Oğuz DİKENELLİ
Ağustos 2018, 69 sayfa

Artan veri yoğunluğu ve çeşitliliğine çözüm olarak ortaya çıkan NoSQL veritabanları beraberinde bir dizi problemi de getirmiştir. Bunlardan en önemlisi performans için farklı depolarda saklanan ilişkili verilerin bütünleştirilmesidir. Bu tezde, farklı NoSQL veritabanları üzerinde saklanan verilerin bütünleştirilmesi problemine odaklanılmıştır. Probleme çözüm olarak kullanılan yaklaşımlar literatürde çoklu depolama sistemleri adıyla anılmaktadır. Tezde bu veri yönetim sistemlerininden örnek alınarak bir prototip gerçekleştirimi yapılmıştır. Prototip tasarlanırken, veri kaynağı dağılımı, heterojenliği ve özerkliği problemine çözüm oluşturabilmek için arabulucu-sarmalayıcı mimariden yararlanılmıştır. Prototip bu mimariye referansla tartışılmaktadır. Ayrıca veri depoları üzerinde sorguların çalıştırılabilmesi için PQL (Polystore Query Language) adında temel operasyonları içeren bir sorgulama dili gerçekleştirimi yapılmıştır. Sonuç olarak dağıtık olmayan yerel depolar üzerinde sorgu işletimi gerçekleştirilmiştir. Sorgu sonucu farklı depolardan dönen sonuçlar, sonuç bütünleştirme işlemi ile bellek üzerinde ortak bir anahtar-değer veri yapısı gerçekleştirimi ile tutulmuştur. Yapılan gerçekleştirim MovieLens veri seti üzerinde yaratılan bir kullanım durumu ile denenmiş ve sonuçlar listelenmiştir.

Anahtar Sözcükler: Çoklu Depo Sistemleri, Arabulucu-Sarmalayıcı Mimari, Büyük Veri Entegrasyonu, NoSQL.

ABSTRACT

DEVELOPMENT OF COMMON QUERY EXECUTION LAYER FOR NOSQL DATABASES

ERDEN, Bahtiyar

MSc in Computer Engineering
Supervisor: Prof. Dr. Oğuz DİKENELLİ
August 2018, 69 pages

The NoSQL databases, which have emerged as a solution to the increased data density and diversity, have also brought a number of problems. The most important of these is the integration of related data which is stored in different repositories for performance. This thesis focuses on the problem of integrating data that stored in different NoSQL databases. Approaches used as probing solutions are referred to in the literature as multistore systems. In the thesis, a prototype has been realized by taking an example from these data management systems. While designing prototypes, the mediator-wrapper architecture has been exploited to solve the problem of data source distribution, heterogeneity and autonomy. The prototype is discussed with reference to this architecture. In addition, a query language that includes basic operations called Polystrore Query Language (PQL) has been implemented so that queries can be run on data stores. As a result, query processing has been performed on non-distributed local repositories. The results that returned from the query processing are held in a common key-value data structure implementation in memory with result integration process. The constructed implementation is tried with a case study created on the MovieLens data set and the results are listed.

Keywords: Multistore Systems, Mediator-Wrapper Architecture, Big Data Integration, NoSQL.

TEŞEKKÜR

Yüksek Lisans öğretim sürecinde her zaman desteğini hisettiğim saygıdeğer danışmanım Prof. Dr. Oğuz Dikenelli başta olmak üzere; tez çalışmasının şekillenmesinde değerli görüş ve önerileri ile katkıda bulunan, hiçbir zaman yardımdan kaçınmayan özverili çalışma arkadaşım ve en büyük destekçim Yük. Müh. Burak Yönyül'e çok teşekkür ederim.

Tüm bu süreçte desteğiyle beni yalnız bırakmayan kadim dostum Batuhan Büyükayhan'a ve en değerli akıl hocam, görüşleriyle yolumu aydınlatan Doç. Dr. Funda Karbek Akarca'ya çok teşekkür ederim.

Hayatım boyunca karşılaştığım her zorlukta beni koşulsuzca destekleyen, öğrenimim için her türlü güçlüğü göğüsleyen ve başarılarımı gururla taşıyan aileme çok teşekkür ederim. Bu bitmeyen bir serüven, bilgisayar bilimleri adına atılmış yalnızca bir adım. Çalışmamı, insanlığı daha ileriye götürmek için doğa ve tarih bilimleri alanında gerekli olan adımları atmasını hayal ettiğim, bu serüvendeki en küçük yol arkadaşım Piraye Akarca'ya adıyorum. Çalışmam adınla yaşasın.

İÇİNDEKİLER

Sayfa

İÇİNDEKİLER (devam)

Sayfa

İÇİNDEKİLER (devam)

Sayfa

ŞEKİLLER DİZİNİ

Şekil Sayfa

ŞEKİLLER DİZİNİ (devam)

Şekil Sayfa

ÇİZELGELER DİZİNİ

Çizelge Sayfa

ALGORİTMALAR DİZİNİ

SİMGELER VE KISALTMALAR DİZİNİ

Kısaltmalar	Açıklamalar
ACID	Atomisite, Tutarlılık, İzolasyon, Devamlılık
ANTLR	Dil Tanımlama İçin Başka Bir Araç
API	Uygulama Geliştirme Arayüzü
BASE	Temel olarak mevcut, Yumuşak Durum, Nihai olarak tutarlı
CAP	Tutarlılık, Kullanılabilirlik, Bölme toleransı
DBMS	Veritabanı Yönetim Sistemi
GAV	Global Olarak Görünüm
HDFS	Hadoop Dağıtık Dosya Sistemi
JSON	Javascript Nesnesi Gösterimi
LAV	Yerel Olarak Görünüm
PODC	Dağıtılmış Hesaplamanın Temelleri
PQL	Çoklu depo Sorgu Dili
QEP	Sorgu Yürütme Planı
SQL	Yapılandırılmış Sorgu Dili
UML	Birleştirilmiş Modelleme Dili

1 GİRİŞ

Yakın geçmişe kadar veri yönetim sistemleri "herkese uygun tek bir çözüm var" prensibine uygun bir gelişim seyri göstermiştir. Ancak veri yoğunluğu ve çeşitliliğinin artması ile birlikte veriyi kullanan uygulamaların, veri kullanım ihtiyaçlarının değişmesi bulut teknolojilerinin ve buna uygun veri yönetim sistemlerinin gelişimini hızlandırmıştır. Veri yönetim prensibinin "herkese uygun tek bir çözüm yok" (Stonebraker and Cetintemel, 2005) olduğunun anlaşılmasıyla birlikte farklı veri tipleri ve görevleri için uzmanlaşmış ve geleneksel ilişkisel veritabanlarının yerine getirebileceğinden daha büyük ve karmaşık görevleri yerine getirebilen veri yönetim çözümleri ortaya çıkmıştır. Yeni veri yönetim teknolojilerinin örnekleri arasında dağıtılmış dosya sistemleri, NoSQL veri depoları ve veri işleme çerçeveleri bulunur.

Veri yönetim çözümlerinin gelişmesi yüksek performası ölçekleyebilecek ve sergileyebilecek uygulamaların oluşturulabilmesi için gerekli olan servislerin ortaya çıkmasını sağlamıştır. Bu aynı zamanda veri depo arayüzlerinin geniş bir çeşitliliğe sahip olmasına ve ortak bir uygulama geliştirme paradigmasının kaybolmasına yol açmıştır. Dolasıyla uygulama geliştiricinin birbirinden farklı arayüzlere sahip veri depolarını kullanabileceği bir uygulama geliştirme bilgisine sahip olmaması ya da bunun zorlu bir süreç olması bulut uygulamalarının geliştirilmesinin önünde bir engel oluşturmuştur.

Farklı modellerde saklanan veriye erişim ve veri bütünleştirme sorunu literatürde çoklu veri depo sistemleri bağlamında tartışılmaktadır (Bondiombouy and Valduriez, 2016). Çoklu veri depo sistemleri üzerindeki çalışmaların çoğu SQL benzeri betik bir dil kullanarak arabulucu-sarmalayıcı mimari üzerinde ortak sorgu işletimi üzerine yoğunlaşmıştır. Arabulucu-samalayıcı mimarisi, veri kaynaklarının dağıtıklık, heterojenlik ve özerklik olmak üzere üç ana özelliğiyle başa çıkılmasını sağlamaktadır. Çoklu veri depo sistemleri karşılaştırmayı kolaylaştırmak için gevşek-bağımlı, sıkı-bağımlı ve melez olmak üzere üç ana kategori altında incelenmektedir.

Bu tez çalışmasının amacı büyük veri entegrasyonu problemine çözüm olarak, çoklu veri depo sistemleri içerisinde bulunan arabulucu-sarmalayıcı mimarisi için bir model önerisi yapmak ve bu öneriye uygun prototip geliştirimini tamamlamaktır. Prototip için gerekli olan öncelikli adım temel operasyonları içeren bir sorgulama dilinin geliştirilmesi olarak belirlenmiştir. Bu sorgulama dili aracılığıyla farklı veri depolarına atılacak olan sorgular aynı arayüz üzerinden gerçekleştirilebilecektir.

Ayrıca sorgu birleştirme ve sonuç bütünleştirme işlemlerinin yapılabilmesi için veri depoları arasındaki ilişkilerin tanımlanması gereklidir. Prototipin geliştirilmesinde belirlenen ikinci adım ise arabulucu-sarmalayıcı mimarinin tasarımı ve gerçekleştirimidir. Arabulucu-sarmalayıcı mimari altta yatan veri depolarına gevşek-bağımlı ancak sorgu çalıştırma bakımından sıkı-bağımlı şekilde dizayn edilmiştir. Geliştirilen bu soyut mimari sayesinde sistemdeki veri deposu sayısı kolayca arttırılabilmektedir. Sorgu dili ve arabulucu-sarmalayıcı mimarinin tasarımı sonrasında ortaya konulan mimari gerçekleştirim, MovieLens veri setinin[1] üzerinde denenmiştir. Sistemin mimari tasarımı ve gerçekleştirimi Bölüm 5'te gösterilmiş olup, çoklu depo sisteminin değerlendirilmesi Bölüm 6'da ortaya koyulmuştur.

Tez çalışmasının kapsamı dışında bırakılan konular sorgu birleştirme ve dağıtık çalıştırma altyapısıdır. İleride yapılacak olan çalışmalar ile sorgu birleştirme algoritmasına (bind join, hash join[2], nested-loop join[3] ve sort-merge join[4]) karar verilip gerçekleştirilmesi hedeflenmektedir. Aynı şekilde çoklu depo sisteminin dağıtık çalışabilmesi için Akka[5] aktör-model ile gerçekleştiriminin yapılması ileriki çalışmalar içerisindedir.

[1]MovieLens veri seti: https://grouplens.org/datasets/movielens/ (Erişim tarihi: 23.06.2018)

[2]Hash Join Algoritması: http://www.dcs.ed.ac.uk/home/tz/phd/thesis/node21.htm (Erişim tarihi: 23.06.2018)

[3]Nested-Loop Join Algoritması: https://en.wikipedia.org/wiki/Nested_loop_join (Erişim tarihi: 23.06.2018)

[4]Sort-Merge Join Algoritması: http://www.dcs.ed.ac.uk/home/tz/phd/thesis/node20.htm (Erişim tarihi: 23.06.2018)

[5]Akka Aktör-Model Çatısı: https://akka.io (Erişim tarihi: 23.06.2018)

2 VERİ DEPOLAMA SİSTEMLERİ

NoSQL sistemler, kullanıcılar, nesneler ve ürünler hakkındaki depolanan verinin hacmindeki bir artışa yanıt olarak geliştirilen ve verilere erişim sıklığı, performans ve işletim ihtiyaçlarını göz önüne alan çok farklı sayıda veritabanı teknolojisi kapsamaktadır. Diğer taraftan ilişkisel veritabanları ne modern uygulamaların karşılaştığı ölçeklenebilirlik ve çeviklik gibi zorluklarla başa çıkabilecek şekilde tasarlanmış ne de bugünkü ucuz depolama maliyetlerinden ve işlem gücünden yaralanmak için inşa edilmiştir. Ancak bu ihtiyaçlara karşılık veremese de ilişkisel veritabanlarının yararlarından bahsetmek ve karşılaştırmalı bir sunum yapmak gereklidir.

2.1 İlişkisel Veritabanları

Veritabanı büyük miktarlarda veri topluluğu (Jatana et al., 2012), veritabanı yönetim sistemleri ise verilerin yönetilmesi, alınması ve saklanması için organize bir yol olarak tanımlanabilmektedir. Veritabanı modelinin güvenilirliği ACID özellikleri (Pritchett, 2008) yardımıyla kontrol edilmektedir. ACID özellikleri ise şu şekildedir:

- *Atomisite*, "her şey ya da hiçbir şey" anlamına gelmektedir. İşlemin herhangi bir kısmı eksik bırakılırsa, tüm işlem başarısız sayılır.
- *Tutarlılık*, bir işlemin öncesinde ve sonrasında bir veritabanının geçerli bir durumda stabil olmasını sağlamak anlamına gelmektedir.
- *İzolasyon*, aynı anda yürütülen birden fazla işlem sırasında, faaliyet halindeki işlemin diğer işlemler tarafından engellenmemesi anlamına gelmektedir.
- *Devamlılık*, bir işlem gerçekleştirildikten sonra durumunu koruması anlamına gelmektedir. Yani bazı hataların ortaya çıkması durumunda, örneğin sistemin çökmesi veya güç kaybı yaşanması durumunda bile veri kalıcı olarak depolanmalıdır.

2.1.1 İlişkisel veritabanlarının avantajları

Bir veritabanının en büyük özelliği büyük miktarlarda veriyi kalıcı olarak saklayabilmesidir (Jatana et al., 2012). Çoğu bilgisayar mimarisinde iki bellek

alanı bulunmaktadır: hızlı bir “ana bellek” ve yavaş ama daha büyük bir “destek deposu”. Ana bellek, depolama alanı bakımından sınırlıdır ve işletim sistemi bir problemle karşılaştığında (güç kaybettiğinde vs.) depoladığı tüm veriyi kaybetme riskine sahiptir. Bu nedenle veriler, genellikle disk olarak görünen destek depolarına yazılmaktadır. Bir destek deposu dosya sistemi ya da veritabanı gibi çok farklı şekillerde organize edilebilmektedir. Ancak veritabanı büyük miktarlarda veriyi depolamak için bir dosya sisteminden daha fazla esneklik sağlamaktadır. Böylece bir uygulama programının bu bilgilerin küçük parçalarına hızlı ve kolay bir şekilde ulaşılmasını sağlamaktadır.

Kurumsal uygulamaların çoğunda veriye aynı anda erişilmekte ve veri üzerinde aynı anda değişiklik yapılmaktadır. Eş zamanlı veri değişikliklerinden oluşabilecek hataları engellemek için bu etkileşimlerin koordine edilmesi gerekmektedir. İlişkisel veritabanları verilere erişimi işlembilgi (transaction) yönetimi aracılığıyla kontrol ederek bu sorunun üstesinden gelmeye yardımcı olmaktadır (Elmasri and Navathe, 2010). İşlembilgi ayrıca hata ayıklamada da önemli bir rol oynamaktadır. İşlembilgi ile değişiklik yapıldığı sırada oluşan hatalar gözlemlenebilmekte ve hata oluştuğu durumlarda işlem geri alınabilmektedir.

Kurumsal uygulamalar, farklı ekipler tarafından yazılan, işleri tamamlamak için işbirliği yapmak üzere birden fazla uygulamanın birlikte çalışmasını gerektiren zengin bir ortamda faaliyet göstermektedir. Uygulamalar genellikle aynı veriyi kullanmaktadır ve bir uygulama tarafından yapılan değişiklik diğerlerince görülebilmelidir. İlişkisel veritabanları bu sorunun üstesinden, birden fazla uygulamanın verilerinin tek bir veritabanında saklandığı, paylaşılan veritabanı entegrasyonu ile gelmektedir.

İlişkisel veritabanlarının başarılı olmasının nedeni yukarıda anlatılanların yanında temel özellikleri standart bir yolla sunabiliyor olmasından kaynaklanmaktadır. Sonuç olarak uygulama geliştiriciler ve veritabanı yöneticileri temel ilişkisel modeli öğrenip bunu birçok projede gerçekleştirebilmektedir. İlişkisel veritabanları arasında farklılıklar olsa da temel mekanizma aynıdır; farklı veritabanı yönetim sistemlerinin SQL lehçeleri benzerdir ve işlembilgi çoğunlukla aynı şekilde çalışmaktadır.

2.1.2 İlişkisel veritabanları ile ilgili sorunlar

İlişkisel veritabanları birçok avantaja sahip olmasının yanısıra bir dizi dezavantaja da sahiptir. Uygulama geliştiriciler açısından en büyük problem

ilişkisel model ile bellek içi veri yapıları arasında bir fark olmasıdır (Jatana et al., 2012). İlişkisel model veriyi, tabloların, satırların ya da daha doğrusu ilişkilerin ve çok-ögeli kayıtların bir yapısı şeklinde organize etmektedir. İlişkisel modelde çok-öğeli (tuple) yapı bir anahtar-değer çiftleri kümesidir ya da ilişki bir dizi çok-öğeli yapıdır. İlişkilere dair bu temel, belirli bir sadelik sağlar ancak aynı zamanda sınırlamalar da getirmektedir (Leavitt, 2010). İlişkisel bir çok-öğeli yapının basit olması gerekir, iç içe geçmiş bir kayıt veya liste gibi herhangi bir yapı içeremezler. Bu sınırlama, ilişkilerden çok daha zengin yapıları barındıran bellek içi veri yapıları için doğru değildir. Dolayısıyla daha zengin veri yapılarını ilişkisel model içerisinde saklamak için bunların ilişkisel bir gösterime dönüştürülmesi gerekmektedir. Aynı şekilde ilişkisel model içerisindeki verinin de tekrar bellek içi yapılara dönüştürülmesi gerekmektedir.

İlişkisel veritabanlarıyla ilgili bir diğer problem de paylaşılan veri tabanı entegrasyonundaki olumsuzluklardır (Sadalage and Fowler, 2012). Birçok uygulamanın entegrasyonu için dizayn edilmiş bir veritabanı tekbir uygulamanın ihtiyacı olandan çok daha karmaşık olmaktadır. Herhangi bir uygulama, veri deposunda bir değişikliğe gittiğinde bunun koordine bir şekilde diğer uygulamalar tarafından da gerçekleştirilmesi gerekmektedir. Ancak farklı uygulamalar farklı yapısal ve performans ihtiyaçlarına sahiptir. Dolayısıyla bir uygulama için gerekli olan bir iyileştirme diğerlerinde problemlere yol açabilmektedir.

İlişkisel veritabanlarının başa çıkamadığı ve veri depolama sistemleri üzerindeki bir değişikliği gerekli kılan en büyük problem ise ölçeklenebilirliktir (Pokorny, 2013; Leavitt, 2010). 90'lı yıllarda internetin çok değişik ölçeklerde kullanılmaya başlamanmasıyla birlikte veri kümelerinde ve kullanıcı sayılarında bir artış meydana gelmiştir (Padhy et al., 2018). Bağlantılar, sosyal ağlar, site etkinlikleri, eşleme verileri vb. gibi verileri içerisinde bulunduran geniş veri kümeleri ve çoğalan veri trafiğini ölçeklendirebilmek, bilgisiyar bilimleri için bir araştırma konusu haline gelmiştir. Ölçekleme iki şekilde mümkün olabilmektedir: Yatay ya da dikey. Dikey ölçekleme daha büyük makineler, daha fazla işlemci gücü, depolama alanı ve bellek anlamına gelmektedir. Yatay ölçekleme ise bir küme (cluster) içerisindeki birçok küçük makineyi kullanarak işi bunlar üzerinde dağıtmak ve küçük makinelerin toplamının oluşturduğu işlemci, depolama ve bellek gücünden yararlanmak anlamına gelmektedir. Tek makine üzerinde çalışmak birçok açıdan avantajlı değildir. Çünkü bilgi işleme ihtiyaçlarının sınırı yoktur. Ancak tek makine üzerinde donanımsal gelişme belirli sınırlara sahiptir. Hata toleransı, esneklik ve maliyet de yatay ölçeklenebilirliği destekleyen unsurlardır. Hata toleransı önemlidir çünkü; tek bir makine üzerinde hata olması durumunda ilerlemek pek mümkün olmazken sistem kümelenmede bu tür arızalara rağmen

devam etmek için dizayn edilebilir. Böylece yüksek güvenilirlik sağlanmış olur. Esneklik ise kümelerin istenildiği zaman genişletilebilir veya daraltılabilir olmasını sağlaması nedeniyle kaynak yönetimini desteklemektedir. Kümele yöntemine geçişle birlikte yeni bir problemle karşı karşıya gelinmiştir; ilişkisel veritabanları kümeler üzerinde çalışacak şekilde tasarlanmamıştır (Sadalage and Fowler, 2012; Stonebraker, 2010; Strauch et al., 2011). Oracle RAC ve Microsoft SQL Server gibi kümelenmiş ilişkisel veritabanları, paylaşılan bir disk alt sistemi üzerinde çalışmaktadırlar. Yüksek düzeyde erişilebilirliğe sahip bir disk alt sistemine yazabilen küme uyumlu bir dosya sistemi kullanırlar. Ancak bu, kümenin hala disk alt sistemini tek bir hata noktası olarak algıladığı anlamına gelmektedir. Diğer yandan ilişkisel veritabanları, farklı veri kümeleri için ayrı sunucular - veritabanı parçalama işlemi (sharding) (Strauch et al., 2011) yaparak - olarak da çalıştırılabilir. Bu durumda yük paylaşılsa da parçalama işlemi, tüm veritabanının her veri biti için hangi veritabanı sunucusu ile konuşulacağını takip eden bir uygulama tarafından kontrol edilmelidir. Ayrıca parçaları birleştiren sorgulama, referans bütünlüğü, işlembilgi ve tutarlılık kontrolleri kaybedilmektedir. Bu gibi teknik problemlerin yanında ilişkisel veritabanlarının lisanslama maliyetleri de her bir sunucu için ayrı olarak hesaplandığından, ilişkisel veritabanları ve kümeler arasındaki bu uyumsuzluk, bazı kuruluşların veri depolama için alternatif bir rota düşünmesine neden olmuştur.

2.2 NoSQL Veri Depoları

"NoSQL" terimi ilk defa 90'lı yılların sonunda Strozzi'nin açık kaynaklı bir ilişkisel veritabanına verdiği isim olarak ortaya çıkmıştır (Sadalage and Fowler, 2012). Ancak bugün bu terim çok daha farklı veri depolama sistemlerine verilen ad olarak kullanılmaktadır. Şimdiki anlamıyla NoSQL, yorumlama ve tanımlama için çok fazla açık kapı bırakan aşırı yüklü bir terimdir. Kimi kaynaklarda "Sadece SQL değil" olarak tanımlanmakta kimi kaynaklarda da "açık-kaynaklı, dağıtık, ilişkisel olmayan"[6] veritabanlarına verilen ortak bir isim olması tartışılsa da genel kabul gören bir tanımı bulunmamaktadır. Bundan dolayı bu tür veritabanı sistemlerini incelerken bunların ortak özellikleri üzerinde durulacaktır.

2.2.1 NoSQL veritabanlarının teorik ilkeleri

CAP Teoremi, BASE Teoremi ve Nihai Tutarlılık Modeli Teoremi NoSQL için üç teorik temel olarak kabul edilmektedir.

[6]NoSQL Toplantıları: http://nosql.eventbrite.com

2.2.1.1 CAP teoremi

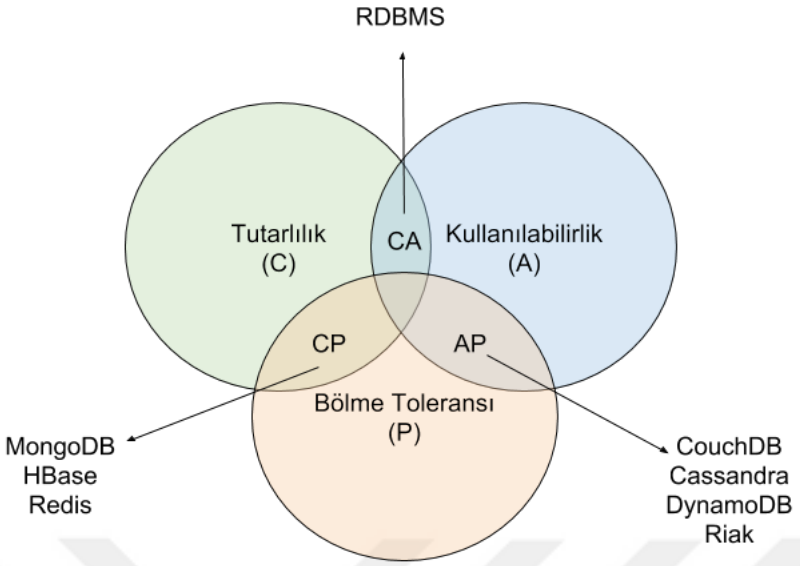
E. Brewer 2000 yılında davetli olduğu PODC'taki[7] konuşmasında CAP teoremini sunmuş (Brewer, 2000), Gilbert ve Lynch bu konuşmadan 2 yıl sonra teoremin geçerliliğini kanıtlamışlardır (Gilbert and Lynch, 2002). Bu teorem, herhangi bir kaynak paylaşımlı sistem için, karşılıklı bağımlılığa sahip üç temel özellik olduğunu belirtmektedir.

- Tutarlılık (C): Her işlemden sonra verilerin veritabanında tutarlı bir durumda bulunması gerektiği anlamına gelmektedir.
- Kullanılabilirlik (A): Veritabanın hiç bir kesinti olmaksızın her zaman kullanılabilir olduğu anlamına gelmektedir. Operasyonun başarılı veya başarısız olmasından bağımsız olarak her isteğin bir cevapla karşılanması gerekmektedir.
- Bölme Toleransı (P): Ağ bölümlenmesi durumunda sistemin çalışmaya devam etmesi gerektiği anlamına gelmektedir.

CAP teoremine göre (Pokorny, 2013; Moniruzzaman and Hossain, 2013; Brewer, 2012; He, 2015; Han et al., 2011), herhangi bir dağıtık veritabanının her üç özelliği (Bkz. Şekil 2.1) de aynı anda yerine getirmesi mümkün değildir. Bu nedenle, dağıtık veritabanları bu üç özellikten en fazla ikisini barındırabilmektedir. Bir dağıtık sistem Bölme Toleransı koşulunu mutlaka sağlamalıdır aksi halde bir veritabanı dağıtık olarak kabul edilmemektedir. Bu nedenle, dağıtık bir veritabanı sisteminin, Tutarlılık ve Kullanılabilirlik arasında bir seçim yapması gerekmektedir.

- **CA Sistemleri:** Veritabanlarının bir kısmı bölme toleransıyla ilgili değildirler ve veri tutarlılığı ile kullanılabilirliğini sağlamak için kopyalama yaklaşımını kullanmaktadırlar. Geleneksel ilişkisel veritabanları yüksek kullanılabilirliği ve tutarlılığı seçerek hem yatay ölçeklenebilirlik hem de iyileştirme için dar bir alan sağlamaktadırlar.
- **CP Sistemleri:** Bu tür veritabanı sistemleri verileri dağıtık düğümlerde saklamaktadır ve veri tutarlılığını sağlamaktadırlar. Ancak veri kullanabilirliği bu tür sistemler için bir problem oluşturmaktadır. Sistem erişilemez olduğunda kullanıcının yazma ve okuma işlemleri başarısız olmaktadır.

[7] Dağıtık Hesaplamanın Temelleri Konferansı: https://www.podc.org



Şekil 2.1: CAP Teoremi

- **AP Sistemleri:** Bu tür sistemler yüksek kullanılabilirliğe ve dağıtık çalışabilme özelliğine sahiptir ancak ağ bölümlere ayrıldığında tutarsızlaşabilmektedir ve kirli okumalar meydana gelmektedir.

2.2.1.2 BASE teoremi

NoSQL veri depoları ACID işlem garantisini sağlamamaktadır. Nispeten daha zayıf bir tutarlılık modeli izlemektedirler. BASE (Pritchett, 2008) "Temel olarak mevcut (BA), Yumuşak durum (S), Nihai olarak tutarlı (E)" şeklinde tanımlanabilmektedir. Temel olarak mevcut, CAP teoremi açısından sistemin kullanılabilirliğini temin etmektedir. Yumuşak durum, herhangi bir girdi verilmese bile sistem durumunun zaman periyodu içerisinde değişebileceğini belirtmektedir. Nihai olarak tutarlı da sistemin zaman periyodu içerisinde bir girdiyle beslenmemesi durumunda zamanla tutarlı hale geleceği anlamını taşımaktadır. BASE teoremi, CAP teoreminin pratikte uygulanmaya başlanmasıyla ortaya çıkmıştır (Wang and Tang, 2012).

BASE, ACID ile taban tabana zıttır. ACID kötümser ise ve her işlemin sonunda sistemin tutarlı hale gelmesini zorluyorsa, BASE iyimserdir ve veritabanı tutarlılığının bir akış durumunda gerçekleşeceğini kabul etmektedir. Bu, BASE ile başa çıkmanın imkansız olmasını düşündürse de gerçekte oldukça yönetilebilirdir ve ACID ile elde edilemeyen ölçeklenebilirlik seviyeleri sağlamaktadır.

2.2.1.3 Nihai tutarlılık modeli

Nihai tutarlılık, paralel programlama alanında kullanılan tutarlılık modellerinden birisidir. Hiçbir değişikliğin gönderilmediği yeterince uzun bir süre neticesinde, tüm güncellemelerin eninde sonunda sistem üzerinde gerçekleştirileceği ve tüm kopyaların tutarlı olacağı anlamına gelmektedir.

Eric Brewer'in CAP teoremini tanıtmasından sonra, Tutarlılık ve Kullanılabilirlik arasındaki farklı ilişkiler incelenmiştir (Brewer, 2012). İstemci ve sunucu tutarlılık için farklı görüşlere sahiptir. İstemci tarafı tutarlılık, gözlemcilerin depolama sistemlerinde bir veri nesnesine yapılan güncellemeleri nasıl ve ne zaman gördükleri ile ilgilidir. Werner Vogels tarafından açıklanan üç tip istemci tarafı tutarlılık bulunmaktadır (Vogels, 2009):

- **Güçlü Tutarlılık:** Güncelleme tamamlandığında, sonraki erişim güncellenmiş değeri döndürür.
- **Zayıf Tutarlılık:** Sistem, sonraki erişimlerin güncellenmiş değeri geri getireceğini garanti etmez.
- **Nihai Tutarlılık:** Zayıf tutarlılığın bir formudur ve depolama sistemi, nesneye yeni güncelleme yapılmadığında, belirli bir süre sonunda tüm erişimlerin son güncellenen değeri döndüreceğini garanti eder.

2.2.2 NoSQL veri depolarının sınıflandırılması

2.2.2.1 Anahtar-değer veri depoları

Veri modeli açısından bakıldığında, anahtar-değer veri depoları NoSQL veritabanlarının en temel ya da basit tipidir. Veritabanı içerisinde her bir eleman, anahtarı ile birlikte elemanın değeri ile depolanmaktadır. Değer sistem içerisinde görünmezdir, veri ancak anahtarı aracılığı ile sorgulanabilmektedir. Bu model, anahtar-değer ikilileri arasında şema oluşturmanın zorunlu olmadığı veritabanlarında, yapısal olmayan ve çok çeşitli veriyi temsil etmede oldukça kullanışlıdır (Inc., 2015). Anahtarı oluşturan "id" sütununa ve değerin tutulduğu "isim" sütununa sahip olan klasik bir ilişkisel veritabanı sistemi ile karşılaştırıldığında; ilişkisel veritabanında "isim" sütunu tipi "karakter dizisi" olan bir depolamayla sınırlıyken, anahtar-değer veritabanlarında bununla ilgili

bir sınırlama yoktur. Burada iç içe geçmiş listeler, kümeler, nesneler vb. gibi karmaşık kayıtlar yer alabilir. Şekil 2.2'de anahtar-değer veri modelinin bir örneği gösterilmiştir.

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

Şekil 2.2: Anahtar-değer veri modeli (Moniruzzaman and Hossain'dan 2013)

Anahtar-değer depoları, API perspektifiyle kullanılmak için en basit NoSQL veri depolarıdır. İstemci anahtar ile değeri alabilir, anahtar ile değer ekleyebilir ya da anahtarı veri deposundan silebilir (Sadalage and Fowler, 2012). Anahtar-değer depoları her zaman birincil anahtar değer erişimini kullandıkları için, genellikle iyi bir performansa sahip olurlar ve kolay bir şekilde ölçeklendirilebilirler.

Modern anahtar-değer veri depoları tutarlılığa göre yüksek ölçeklenebilirliği tercih etmektedirler (Nayak et al., 2013). Bu nedenle, toplama ve birleştirme işlemleri gibi ad-hoc sorgulama ve analiz işlemleri ihmal edilmiştir. Yüksek eşzamanlılık, hızlı aramalar ve büyük veri depolama bu tür veritabanlarının avantajları arasında yer alırken, şemasız oluşu nedeniyle veri üzerinde özel görünümler oluşturamaması eksiklikleri arasında yer alabilir.

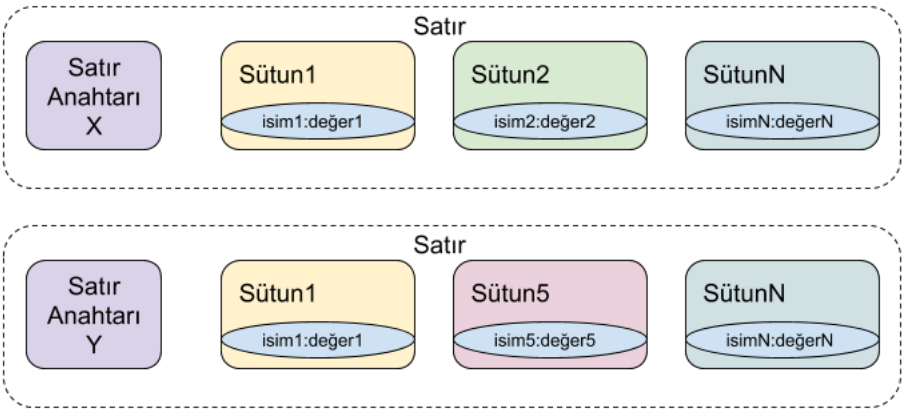
Veri depolama şekline göre bu tip veritabanları geçici, kalıcı ve hibrit kategoriler altında incelenebilmektedir (He, 2015). Geçici veritabanı, veriyi sabit diske yazmadığı için okuma ve yazma işlemlerini hızlandırabilmektedir. Ancak sistem bir hatayla karşılaştığında veriler kaybedilebilir. Kalıcı veritabanları verileri sabit diske kaydetmektedir. Performansı, verileri işlerken sabit diskte G/Ç işlemlerini gerçekleştirmesi gerektiğinden geçici veritabanına göre daha düşüktür. Ancak en büyük avantajı verilerin, herhangi bir sistem hatasıyla karşılaşıldığında kaybolmamasıdır. Hibrit veritabanı geçici ve kalıcı anahtar-değer depolamasının avantajlarını birleştirmektedir. Önce verileri dahili depoya kaydetmektedir ve ardından belirtilen koşullar yerine getirildiğinde girişi sabit diske yazmaktadır. Bu, geçici veritabanının veri işlem hızının kaynağını oluşturmaktadır. Verileri sabit

diske yazarak da veri devamlılığı garanti edilmektedir.

2.2.2.2 Geniş-sütun veri depoları

Geniş sütun depoları ya da diğer bir adıyla sütun aile depoları, veriyi depolamak için aralıklı, dağıtık çok boyutlu sıralı bir harita kullanır. Her bir sütunda depolanan kayıt sayısı değişebilir. Sütunlar, sütun ailelerine erişim için birlikte gruplandırılabilirler ya da sütunlar birden fazla sütun ailesi üzerine yayılmış olabilir. Veriler sütun ailesine ait olan birincil anahtarlar ile alınır (Inc., 2015). Veri modeli Şekil 2.3'te gösterilmiştir.

Anahtar-değer depoları ve doküman veritabanları satır odaklıdır. Bunun anlamı bu tür veritabanlarının bir ya da daha fazla kıstasa göre belirlenen bütün veriyi getirme eğiliminde olmalarıdır. Ancak bazen uygulama bütün doküman yerine alanların alt kümesinin getirilmesine ihtiyaç duyar. Sütun-aile veritabanları, her aile sütunu verinin haritası olacak şekilde, veriyi değerlere haritalanmış anahtarlar olarak ve değerleri birden fazla sütun ailesi olacak şekilde depolarlar. Aile-sütunları, genellikle beraber erişilen bir grup ilişkili verilerdir (Sadalage and Fowler, 2012). Veriler tablolarda değil ancak büyük ölçüde dağıtılmış mimarilerde, G/Ç etkinliği ile hızlı bir şekilde toplayabilecek bir biçimde saklanmaktadır. Veri depolamada yüksek ölçeklenebilirlik sunmaktadırlar (Nayak et al., 2013).



Şekil 2.3: Geniş-sütun veri modeli

Geniş-sütun depoları o kadar ölçeklenebilirdir ki, veri boyutundaki artış, işlem hızının azalmasına neden olmamaktadır(He, 2015). Bu yüzden büyük miktarda veriyi işlemek için çok uygundurlar.

2.2.2.3 Doküman veri depoları

İlişkisel veritabanları verileri satırlar ve sütunlar içerisinde depolarlar. Bunun aksine doküman veritabanları verileri doküman içerisinde depolarlar. Bu dokümanlar, geliştiriciler tarafından popüler olan JSON gibi yapısal bir format kullanırlar (Şekil 2.4). Dokümanlar, nesne yönelimli programlama ile uyumlu verileri modellemek için sezgisel ve doğal bir yol sağlarlar. Çünkü her doküman etkin bir şekilde bir nesnedir. Dokümanlar, bir veya birden fazla alan içerir ve her bir alanın içerisinde örneğin string, date, binary ya da array gibi tipli değerler yer alabilir. Bir kaydı, birbirleriyle ikincil anahtarlarla bağlanmış tablolar ve sütunlara yayılarak saklamaktansa, her bir kayıt ve onunla ilişkili veriler tek bir dokümanın içerisinde beraber saklanırlar. Bu veri erişimini kolaylaştırır ve bir çok durumda maliyetli birleştirme işlemlerine ve karmaşık çok-kayıtlı işlemlere olan ihtiyacı ortadan kaldırır. Doküman veritabanında, şema kavramı dinamiktir: her doküman birbirlerinden farklı alanları içerebilir. Bu esneklik yapılandırılmamış ve polimorfik veri modellemesinde özellikle yararlı olabilir (Inc., 2015). Aynı zamanda, bu özellik uygulamanın geliştirim sırasında -örneğin yeni bir alan ekleme gibi- evriltilebilmesine kolaylık sağlar. Ayrıca, doküman veritabanları genellikle geliştiricilerin ilişkisel veritabanlarından beklediği sorgu sağlamlığını sağlayabilirler. Özel olarak, veriler, dokümandaki alanların herhangi bir kombinasyonuyla sorgulanabilirler (Moniruzzaman and Hossain, 2013).

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

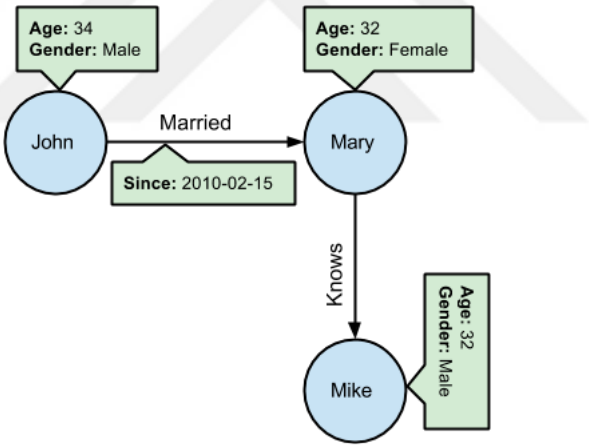
Şekil 2.4: Doküman veri modeli (Microsoft'tan 2017)

2.2.2.4 Çizge veri depoları

Çizge veri depoları, veriyi düğümler, kenarlar ve özellikler gibi çizge yapılarını kullanarak temsil etmektedir (Inc., 2015). Özünde veri, özel elemanlar

arasındaki ilişkiler ağı olarak modellenir. Düğümler bir uygulamadaki nesne örnekleri olabilir. İlişkiler de özellikleri olan kenarlar olarak adlandırılmaktadır. Kenarlar yönlü bir öneme sahiptir, düğümler, düğümler arasında ilginç desenleri bulabilmek için ilişkiler tarafından organize edilir (Şekil 2.5). Çizge modelini anlamak biraz zaman alsa da belirli bir sınıf için olan sorgularda yararlı olabilmektedir. Uygulama içerisindeki varlıklar arasındaki ilişkileri modellemek ve bu varlıklar arasında gezinmek çizge veritabanının en önemli özelliğidir (Sadalage and Fowler, 2012). Çizge organizasyonu, verinin bir kere saklanmasına ve daha sonra ilişkilere bağlı olarak birden fazla yolla yorumlanmasına olanak sağlar.

Çizgeyi sorgulamak aynı zamanda çizge üzerinde gezinmek anlamına da gelmektedir. Çizge veritabanlarının avantajı, düğümleri ya da kenarları değiştirmeden çizgede dolaşma ihtiyaçlarımızı değiştirebiliyor olmamızdır (Nayak et al., 2013). Çizge veritabanlarında, bağlantıları ya da ilişkileri dolaşmak çok hızlı gerçekleşen bir işlemdir. Düğümler arasındaki ilişkiler sorgulama zamanında hesaplanmazlar, düğümler zaten ilişkiler olarak tutulur. İlişkileri dolaşmak onları her sorgu için hesaplamaktan daha hızlıdır (Sadalage and Fowler, 2012).



Şekil 2.5: Çizge veri modeli (Shilov'dan 2014)

2.2.3 NoSQL veri depolarının avantajları

NoSQL veritabanlarının ana avantajları şu başlıklar altında incelenebilir: Hızlı veri yazma ve okuma işlemleri, büyük veri deposu olma işlevleri, kolay genişletilebilir olmaları ve düşük maliyetli olmaları (Han et al., 2011). NoSQL veritabanları veri modellerinin farklılığına göre özelleşmiş veritabanlarıdır. Anahtar-değer veri modelinde değere ulaşmak $O(1)$ zaman karmaşıklığında olmaktadır. Doküman veritabanlarında birlikte kullanılacak veriler dokümanlar halinde depolandığı için ilişkisel veritabanlarında olduğu gibi karmaşık birleştirme sorgularına ihtiyaç duymazlar ve böylece sorgu performansları ilişkisel

veritabanlarına göre daha iyi olmaktadır. Aynı şekilde çizge veritabanında karmaşık sorgular ilişkisel veritabanına oranla daha iyi çalışmaktadır. Ayrıca NoSQL veritabanları ACID'i yerel olarak desteklemediği için işlembilgi yönetimi maliyetlerini içermemektedir. Bu yüzden okuma yazma hızı açısından ilişkisel veritabanlarına göre daha performanslı çalışmaktadırlar.

İlişkisel veritabanlarının aksine NoSQL veritabanları kümeler üzerinde kolayca ölçeklenebilmektedir. Böylece dağıtıldıkları makinelerin işlem güçlerinden, depolama alanlarından ve belleklerinden yararlanmaktadırlar. Bu yolla büyük veri depolama ve analitik için tercih edilen veritabanı sistemi haline gelmişlerdir (Moniruzzaman and Hossain, 2013). Ayrıca NoSQL veritabanları açık kaynaklı yazımlar olduklarından dağıtık çalışma maliyeti tek bir makine üzerinde çalışmadan daha az olmaktadır.

2.2.4 NoSQL veri depolarının dezavantajları

NoSQL veritabanları SQL ile çalışmadığı için, basit görevler için hızlı olabilecek, ancak başkaları için zaman alıcı olabilecek manuel sorgulama programlarına ihtiyaç duyarlar (Leavitt, 2010). Ayrıca NoSQL veritabanlarını sorgulayabilecek ortak bir sorgu dili olmadığı için veri entegrasyonu senaryolarında büyük problemlerle karşılaşılmaktadır.

İlişkisel veritabanları ACID'i desteklerken, NoSQL veritabanları desteklememektedir (Leavitt, 2010). Dolayısıyla, NoSQL veritabanları, ACID'in sağladığı güvenilirlik derecesini yerel olarak sunamamaktadırlar. Kullanıcılar NoSQL veritabanları üzerinde ACID kısıtlamalarını istiyorlar ise ek geliştirme yapmaları gerekmektedir.

NoSQL veritabanları, ACID işlemlerini yerel olarak desteklemediğinden, manuel destek sağlanmadığı sürece tutarlılıktan ödün verebilmektedirler (Leavitt, 2010). Tutarlılığın sağlanamamasına karşın bu tür veritabanları daha iyi performans ve ölçeklenebilirlikle çalışabilmektedirler. Ancak bu durum bazı uygulama ve işlem türleri için bir sorun teşkil etmektedir. Ayrıca bir veritabanı sistemi CAP teoreminde anlatıldığı üzere her üç özelliği de barındıramayacaklarından tutarlılık için geçerli olan bu sorun kullanabilirlik ve dağıtık çalışabilirlik özellikleri için de geçerlidir.

NoSQL veritabanlarının çoğu, arabellek havuzu ve çok iş parçacığını gerçekleştiren disk tabanlı bir yapıya sahip olduğu için performans ek yüküne ek olarak arabellek yönetimi ve kilitleme özelliklerini de beraberinde getirmektedir

(Jatana et al., 2012).

NoSQL veritabanları gelişimini tamamlamamış ve yaygın olarak kullanılmamaktadır (Nayak et al., 2013). Dolayısıyla uygulama geliştiriciler açısından bununla başa çıkmak zor olabilmektedir. Ayrıca standart bir arayüz sunmamaları uygulama geliştiriciler açısından bir diğer problem kaynağıdır (Nayak et al., 2013).

3 ÇOKLU VERİ DEPOLAMA SİSTEMLERİ

Birim zamanda kaynaklar arasında aktarılan veri miktarının artması, verinin hacimce büyümesi ve veri çeşitliliğinin artmasıyla birlikte ilişkisel veri modeli yaklaşımıyla ölçeklenemeyen verinin; depolanma, ölçeklenme ve sorgulanma problemlerine bir çözüm olarak sunulan çoklu veri depolama sistemleri "Her bir soruna uygun tek bir çözüm yoktur" mottosu ile hareket etmektedir. Buna göre farklı veri saklama ihtiyaçlarını yönetebilmek için farklı veri saklama teknolojileri kullanılmalıdır.

Veritabanı sistemleri, temelinde bağlı bulundukları şemaya göre; ilişkisel, anahtar-değer, geniş-sütun, doküman ve çizge olmak üzere beş sınıf altında toplanmaktadır. Farklı tipte şemaya sahip depoların birlikte ve bütünleşik olarak yönetilmesi veritabanı literatüründe çoklu depo olarak ifade edilmektedir.

Çoklu veri depolama sistemleri farklı veri yönetim sistemlerini, sorunsuz ve şeffaf bir biçimde entegre ederek kullanıcılara bütünleşik bir arayüzle sunmalıdır (Dziedzic et al., 2016). BigDAWG çalışmasında bir çoklu veri depolama sisteminin özellikleri (Gadepally et al., 2017) şu şekilde tanımlanmıştır:

- Çoklu veri depolama sistemi **birden fazla veri deposundan** oluşmalıdır. Ancak, çoklu veri depoları, tek bir sorgu motoru ile çalışan ve bir depolama altyapısının kopyalanmış örneklerinden oluşan dağıtık DBMS ile karıştırılmamalıdır. Çoklu veri deposunun belirliyici özelliklerinden biri, çoklu depolama motorlarının ayrı olması ve kendi sorgu motorları aracılığıyla ayrı ayrı erişilebilmesidir.

- Depolama motorları bir çoklu veri depolama sisteminde **heterojen** olmalıdır.

- Depolama motorları **entegre** çalışmalıdır.

Yukarıda da belirtildiği üzere çoklu depolama sistemleri birden fazla veri deposundan oluşmaktadır. Uzun süredir heterojen veri kaynaklarına erişim problemi çoklu veritabanı sistemleri bağlamında incelenmektedir (Özsu and Valduriez, 2011). Çoklu veritabanı üzerinde sorgu işletme çalışmalarının çoğu SQL benzeri bildirimsel bir dil kullanılarak arabulucu-sarmalayıcı mimarisi bağlamında gerçekleştirilmiştir.

3.1 Arabulucu-Sarmalayıcı Mimari

Bir çoklu veritabanı sistemi bilgisayar ağı üzerinde dağıtılmış çoklu, heterojen veri kaynakları kümesine şeffaf bir erişim sağlamaktadır. Dağıtık ve heterojen olmanın yanı sıra veri kaynakları özerk olabilmektedir. Yani çoklu veritabanı sistemi dışından kontrol edilebilmekte ve yönetilebilmektedirler. Arabulucu-sarmalayıcı mimari, veri kaynaklarının bu üç ana özelliği ile ilgilenmeye izin vermektedir. Arabulucu (Wiederhold, 1992) veri kaynağı dağılımı ile ilgilenirken sarmalayıcılar veri kaynağı heterojenliği ve özerkliği ile ilgilenmektedirler (Bondiombouy and Valduriez, 2016). Bu, arabulucu ve sarmalayıcılar arasında ortak bir dil kullanılarak elde edilmektedir ve sarmalayıcılar tarafından veri kaynağı diline çevrim işlemi yapılmaktadır.

Her veri kaynağının, kaynak şeması, veri ve sorgu işleme kapasiteleri hakkında bilgi veren bir sarmalayıcısı bulunmaktadır. Veri kaynaklarının heterojen doğasını ele almak için sarmalayıcılar, arabulucundan gelen ortak bir sorgu diliyle ifade edilen sorguyu, kaynağın sorgu diline dönüştürmektedir. Bir sarmalayıcı, belirli bir sunucuya uygun sorguları tercüme etme ve arabulucuya uygun cevapları (verileri) yeniden biçimlendirme işlevini desteklemektedir.

Veri bütünleştirme konusunun temel uğrağını farklı veri kaynaklarından gelen verinin birleştirilmiş bir görünümde sunulması oluşturmaktadır. Konunun teorik temelleri üzerinde duran Lenzerini (2002) temel olarak; bir veri bütünleştirme uygulamasının modellenmesi, veri bütünleştirmede sorgu çalıştırma, tutarsız veri kaynaklarıyla başa çıkma ve sorgular üzerinde çıkarsama konuları üzerinde durmuştur. Veri bütünleştirme sistemleri global şema ve kaynaklar kümesi olmak üzere iki temel bileşeni baz alan bir mimari üzerine kurulmaktadır. Yazılım mimarisi açısından yerel olarak görünüm (LAV) ve global olarak görünüm (GAV) olmak üzere iki ana yaklaşım bulunmaktadır. LAV yaklaşımında global şema, kaynaklardan bağımsız olarak tanımlanır ve global şema ile kaynaklar arasındaki ilişkiler ise her bir kaynağı global şemanın görüntüsü olarak tanımlayarak kurulur. LAV yaklaşımı global şemanın sabit ve iyi tanımlanmış olduğu durumlarda avantajlıdır. Global şema değişmediğinden yeni bir kaynak sisteme kolayca eklenebilir ve sistem kolayca genişletilebilir. Yeni kaynağın diğer kaynaklar ile bağlantısı global şema üzerinde tanımlanır. Sorgu işletimi görünümler üzerinden eşlemelerin tahmin edilmesi ile yapılır ve tam sonuç garanti edilmez. Verinin bütünleştirilmesi global şemadaki eşlemelere göre sorgu işlemcisi tarafından ele alınır. GAV yaklaşımında ise global şema, kaynaklar cinsinden ifade edilerek oluşturulur. GAV yaklaşımı kaynakların sabit olduğu ve sorgu çalıştırımlarının

yoğun olduğu veri bütünleştirme sisteminde etkilidir. Sorgu işletimi LAV yaklaşımına göre daha basittir. Global şema ile yerel şemalar arasındaki sorgu ve veri dönüşümü sarmalayıcılar aracılığıyla gerçekleştirilir. Her alt sorgu ilişkili olduğu kaynak üzerinde işletilip sonuçları sarmalayıcılar tarafından dönüştürüldükten sonra birleştirilerek ana sorgu sonucu elde edilir. Sorgu, sarmalayıcılar tarafından ele alındığından dönüşüm önceden bellidir ve tam sonuç garanti edilir. Ancak sisteme yeni kaynak eklemek zordur, çünkü yeni eklenen şema için veriyi ve sorguyu dönüştürecek bir sarmalayıcı yazılması gereklidir.

Arabulucu, sarmalayıcılar tarafından sağlanan bilgileri mevcut tüm verilerin birleştirilmiş bir görünümünde sunmaktadır. Bu birleşik görünüm yukarıda anlatıldığı üzere iki biçimde olabilmektedir: LAV ve GAV. Diğer yandan arabulucuların ana işlevselliğini, birden fazla veri kaynağına tekdüzen erişim sağlamak ve veri kaynaklarına erişmek için sarmalayıcıları kullanarak sorgu ayrıştırma ve çalıştırma işlemini gerçekleştirme oluşturmaktadır.

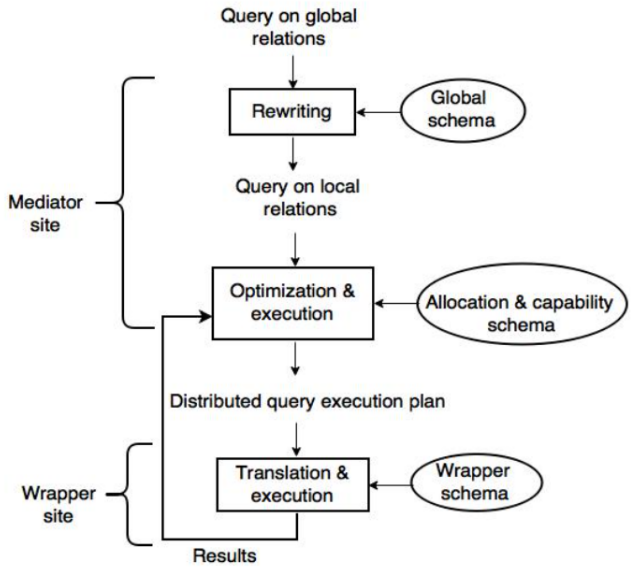
3.1.1 Çoklu veritabanı sorgu işletim mimarisi

Çoklu veritabanı sorgu işletim mimarisinde üç ana katman bulunmaktadır. Bunlardan ilkini sorgu yeniden yazma katmanı oluşturken ikincisini sorgu eniyileştirme ve çalıştırma ile üçüncüsünü dönüştürme ve çalıştırma oluşturmaktadır. Sorgu işletim mimarisine gelen girdinin, bildirimsel bir dille ifade edilen global şema üzerinde tanımlı ilişkilere ilişkin bir sorgu olduğu kabul edilmektedir.

İlk iki katman, giriş sorgusunu optimize edilmiş bir sorgu yürütme planına (QEP) eşler. Sorgu yeniden yazma, sorgu optimizasyonu ve bazı sorgu yürütme işlevlerini yerine getirirler. İlk iki katman, arabulucu tarafından gerçekleştirilir ve genel katalogda saklanan meta bilgileri (global şema, veri kaynağı konumu, maliyet bilgisi, vb.) kullanır. Sorgu yeniden yazma, genel şema kullanarak giriş sorgusunu yerel ilişkilerdeki bir sorguya yeniden yazar. Böylece, global şema, görünüm tanımlarını (yani global ilişkiler ve veri kaynaklarında saklanan lokal ilişkiler arasındaki GAV veya LAV eşleştirmelerini) sağlar ve görünümler kullanılarak sorgu yeniden yazılır.

İkinci katman, ilişkilerin yerini ve sarıcılar tarafından dışa aktarılan veri kaynaklarının farklı sorgu işleme özelliklerini dikkate alarak dağıtılmış sorgu optimizasyonunu ve bazı yürütme işlemlerini gerçekleştirir. Bu katman tarafından üretilen dağıtılmış QEP, alt sorgular içinde veri kaynakları ve

sarmalayıcılar tarafından gerçekleştirilebilen işlemleri içerir. Bununla birlikte, çoklu veritabanı sistemlerinde homojenlik olmaması (ör. Bazı veri kaynakları, cevap vermede beklenmedik uzun gecikmelere sahip olabilir), dinamik sorgu optimizasyonunu önemli hale getirir. Dinamik optimizasyon durumunda, bu katmanın sonraki işlemlemlerine yürütme işlemi sonrasında bir sonraki katman tarafından çağrılar olabilmektedir. Son olarak, bu katman farklı sarmalayıcılardan gelen sonuçları kullanıcı sorgusuna birleşik bir cevap sağlamak için birleştirir. Bu sarmalayıcılardan gelen veriler üzerinde bazı işlemleri yürütme yeteneğini gerektirir.



Şekil 3.1: Çoklu veritabanı sorgu işletim mimarisi (Bondiombouy and Valduriez'den 2016)

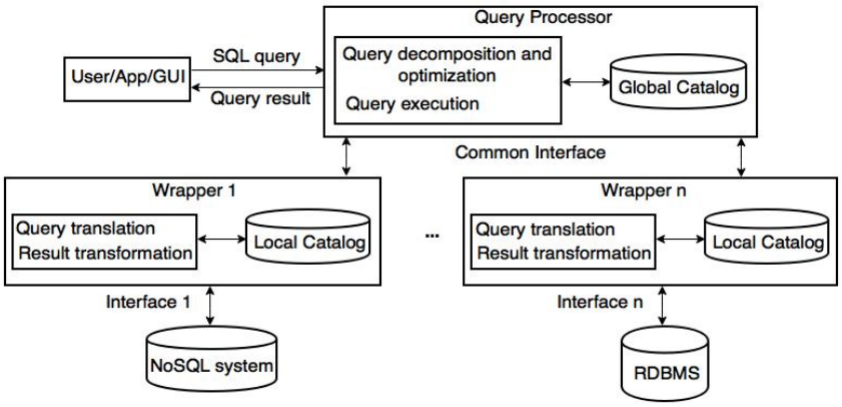
Üçüncü katman, sarmalayıcıları kullanarak sorgu çevirisi ve yürütme işlemini gerçekleştirir. Daha sonra sonuçları, farklı sarmalayıcılardan ve alt sorgu işletiminden gelen sonuçlara entegre edebilen arabulucuya döndürür. Her sarmalayıcı, ortak bir dilde ifade edilen girdi alt-sorgusunun (QEP'in bir alt-kümesinin) veri kaynağının diline çevrilmesini kolaylaştırmak için yerel şema ve eşleme bilgisini içeren bir sarmalayıcı şemasını muhafaza eder. Alt sorgu çevrildikten sonra, veri kaynağı tarafından yürütülür ve yerel sonuç ortak biçimde geri çevrilir (Bondiombouy and Valduriez, 2016). Mimari Şekil 3.1'de görülmektedir.

3.2 Veri Depolarına Bağlanma Seviyesine Göre Çoklu Depoların Sınıflandırılması

Çoklu depo sistemleri üzerinde çalıştıkları veri depolarına bağlanma seviyesine göre gevşek-bağımlı, sıkı-bağımlı ve hibrit sistemler olmak üzere üç ana kategori altında incelenmektedir (Bondiombouy and Valduriez, 2016). Bu ana kategorilerde yer alan sistemler de belli başlı alt birimlerden meydana gelmektedir. İlerleyen alt bölümlerde her bir kategorinin mimari yapısı ve içerisinde bulunan modüller tanıtılacaktır.

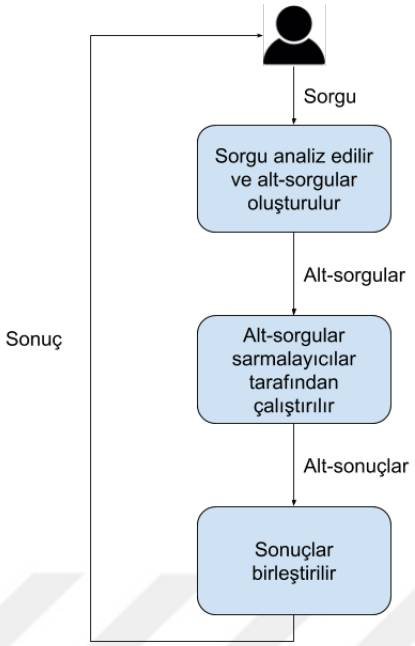
3.2.1 Gevşek-bağımlı çoklu depo sistemleri

Gevşek-bağımlı çoklu depo sistemleri, veri depoların özerkliğiyle başa çıkabilmektedir. Veri depoları yerel olarak kontrol edilebilir ve diğer uygulamalar tarafından erişilebilirdir. Şekil 3.2'de de gösterildiği gibi çeşitli veri depoları ile (örneğin RDBMS ve NoSQL) arabulucu-sarmalayıcı mimariyi gerçekleştirirler. Buradaki veri depoları özerktir. Bu bağlanma seviyesinin avantajı içerisinde çok fazla sayıda veri deposunu barındırabilmesinden gelmektedir.



Şekil 3.2: Gevşek-bağımlı çoklu depo sistemleri (Bondiombouy and Valduriez'den 2016)

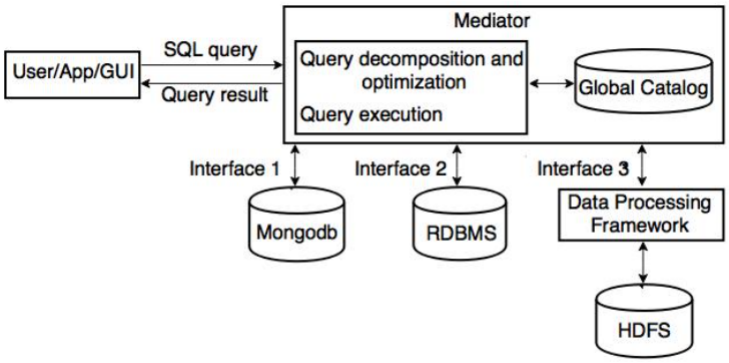
Gevşek-bağımlı çoklu depo sistemlerinin iki ana modülü bulunmaktadır: Bir sorgu işlemcisi ve her bir veri deposuna özgü sarmalayıcı. Sorgu işlemcisi, veri depolarının bir kataloğuna sahiptir ve her sarmalayıcı, kendi veri deposunun yerel bir kataloğuna sahiptir. Tipik bir sorgu işletimi Şekil 3.3'te gösterilmiştir.



Şekil 3.3: Gevşek-bağlı mimari de sorgu işletimi

3.2.2 Sıkı-bağımlı çoklu depo sistemleri

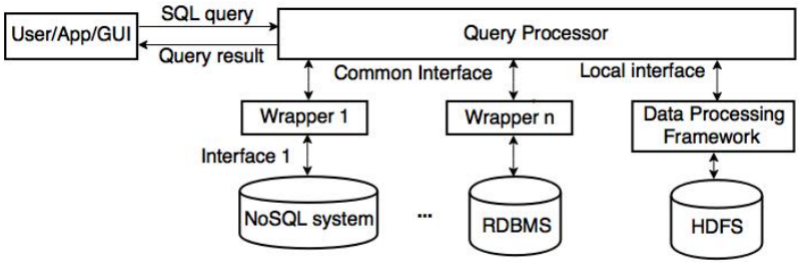
Sıkı-bağımlı çoklu depo sistemleri, -büyük- veri analizi için yapılandırılmış ve yapılandırılmamış verilerin verimli bir şekilde sorgulanmasını amaçlamaktadır. Ayrıca HDFS ve RDBMS verilerinin entegrasyonu gibi belirli bir hedefleri olabilmektedir. Bu tür sistemler paylaşımsız bir küme üzerinde performans için özerkliği göz ardı ederler. Bu nedenle veri depolarına erişim ancak çoklu depo sistemi üzerinden veri depolarının yerel API'leri ile sağlanmaktadır. Sistemin mimarisi Şekil 3.4'te gösterilmiştir.



Şekil 3.4: Sıkı-bağımlı çoklu depo sistemleri (Bondiombouy and Valduriez'den 2016)

Gevşek-bağımlı sistemler gibi, yapılandırılmış ve yapılandırılmamış verilerin sorgulanması için tek bir dil sağlamaktadırlar. Ancak, sorgu işlemcisi doğrudan

yerel depo arabirimlerini kullanmaktadır. HDFS[8] üzerindeki verilere erişim durumunda ise MapReduce[9] veya Spark[10] gibi veri işleme çerçevelerini arabirim olarak kullanmaktadırlar. Böylece sorgu yürütme işlemi sırasında, sorgu işlemcisi doğrudan veri depolarına erişir. Bu mimari veri depoları arasında performanslı veri hareketine izin verirken arayüzlenebilen veri depolarının sayısını kısıtlamaktadır.



Şekil 3.5: Hibrit çoklu depo sistemleri (Bondiombouy and Valduriez'den 2016)

3.2.3 Hibrit Sistemler

Hibrit sistemler, gevşek-bağlımlı sistemlerin birden fazla veri deposuna erişebilme gibi avantajları ile sıkı-bağımlı sistemlerin etkin erişim için veri depolarına onların yerel arayüzleriyle erişebilme gibi avantajlarını birleştirmeyi amaçlamaktadır. Şekil 3.5'te gösterildiği gibi mimari, sorgu işlemcisi HDFS'e doğrudan erişmek için MapReduce ve Spark gibi veri işleme çerçevelerini kullanırken aynı zamanda arabulucu-sarmalayıcı mimarisini takip etmektedir.

[8]Hadoop Dağıtık Dosya Sistemi: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[9]MapReduce Yazılım Çatısı: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

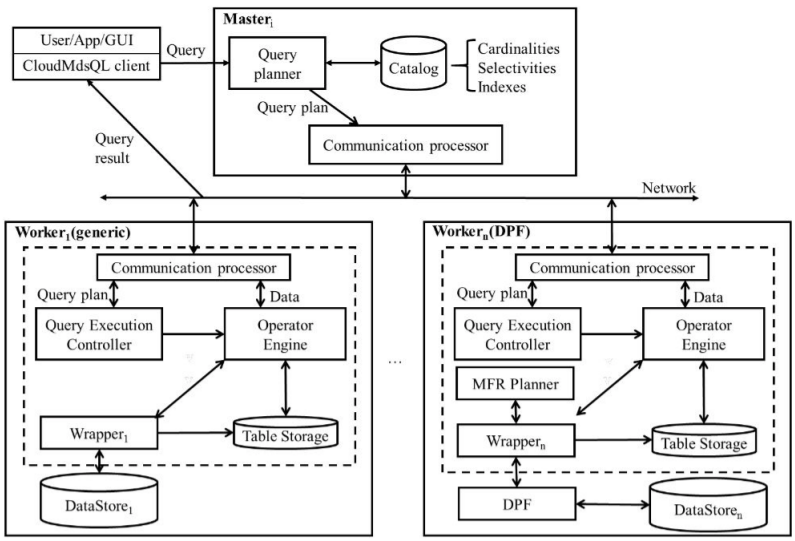
[10]Spark Birleşik Analitik Motoru: https://spark.apache.org

4 ÖNCEKİ ÇALIŞMALAR

Veri entegrasyonu sorunu, araştırma grupları için çözümü öncelikli problemler arasında yer almaktadır. Dolayısıyla bulut tabanlı çoklu depolama sistemleri ile ilgili çeşitli araç ve yayın çalışmaları üretilmektedir. Bu bölümde geliştirilen araç ve önerilen mimari yapılar incelenecektir.

4.1 CloudMdsQL

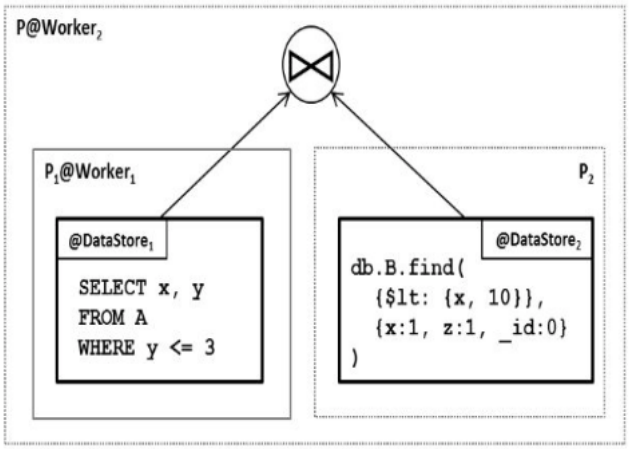
Kolev vd. (Kolev et al., 2016a) çalışmalarında bir çoklu depo sistemi olan CloudMDSQL'in (Bondiombouy et al., 2015; Bondiombouy et al., 2016; Bondiombouy and Valduriez, 2016; Kolev et al., 2016d; Kolev et al., 2016a) tasarımını ve gerçekleştirimini ortaya koymuşlardır. Bu çalışmada tasarlanan sorgu dili ve bu dile özel geliştirilen sorgu motoru tüm detayları ve işleyişiyle ele alınmıştır. Öncelikle veri modellerine ve sorgu dilindeki temel yapılara değinilmiştir. Daha sonra sorgu motorunun yapısı ve temel bileşenleri anlatılmıştır. Dağıtık halde sahip-köle (master-slave) yapıda çalışan (Şekil 4.1) sorgu motorundaki düğümler veri ve sorgu planı değiş tokuşu yapabilmektedir ve sorgu işletimi yöntemlerinden bağlı bileşimi (bind-join) esas almaktadır. Her düğüm (node) bir Sahip (master) ve bir İşçi'den (worker) oluşmakta ve birbirleriyle bir iletişim işlemcisi aracılığıyla haberleşmektedirler. Düğümler sorgu planlama için bir Sahip, sorgu işletimi için ise bir ya da daha fazla İşçi içermektedirler.



Şekil 4.1: CloudMdsQL sorgu motoru mimarisi (Kolev et al.'dan 2016)

Sahip girdi olarak bir sorgu alır ve çıktı olarak da bir sorgu planı oluşturur. Sorgu planı oluşturulurken, veri kümesi bilgilerinin bulunduğu bir üst veri

bilgisinden ya da maliyet modelinden faydalanılır. Yapılan eniyileştirmeler (selection push-down, bind join vb.) sonucu oluşturulan sorgu planının çalışabilirliği, yerel depoların özellikleri dikkate alınarak sarmalayıcılar aracılığıyla doğrulanır. Daha sonra bu sorgu planı master tarafından seçilen bir İşçi'ye çalıştırılmak üzere iletilir. Sahip tarafından üretilen örnek bir sorgu planı Şekil 4.2'de görülmektedir.



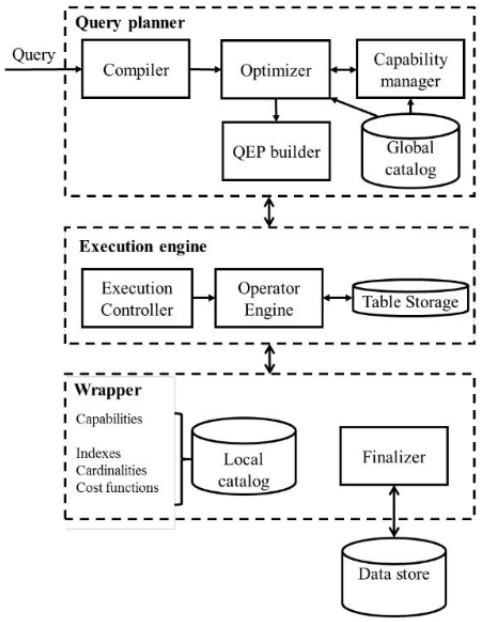
Şekil 4.2: CloudMdsQL örnek sorgu planı (Kolev et al.'dan 2016)

İşçiler sorguda verilen veri depoları üzerinde sorgu planını işletmek için birbirleriyle işbirliği ve iletişim halindedirler. Belirli bir İşçi düğümü sorgu planının işletilmesi sorumluluğunu alır ve diğer İşçi düğümleriyle iletişim kurarak alt sorgu sonuçlarını bütünleştirip ana sonucun oluşturulmasını sağlar. Sistemin genel gerçekleştirim mimarisi Şekil 4.3'teki gibi ortaya konmuştur.

Sorgu dilinin özgünlüğü ise hem SQL sorgularını hem de veri depolarına özgü sorguları bir arada çalıştırabilmesidir.

Kolev vd. (Kolev et al., 2016b; Kolev et al., 2016c) CloudMdsQL'in örneklenmesi için iki farklı senaryo ortaya koymuşlardır: Pazarlama şirketleri için bir sosyal ağ analiz aracı ve bir bibliyografik öneri sistemi. İlk kullanım örneği, toplulukları bir sosyal ağda, belli başlı konular için, en iyi etkileyicileriyle bulmayı amaçlar. Pazarlama şirketleri, belirli bir markanın kalitesi konusunda ikna etmek için ihtiyaç duydukları insanları keşfetmeye ilgi duymaktadırlar. Bu kullanım örneğinin veri kümesi, Twitter'ın bir örneğidir, ancak Facebook veya bloglar gibi diğer sosyal ağlarla çalışmayı sağlar. Uygulama, gerçek zamanlı olarak bir dizi başlığın bir Twitter dinleyicisini çalıştırır; aldığı her tweet için veritabanını değiştirir.

Bu uygulamanın şeması, metin öğelerini saklamak için *Belge* adı verilen genel bir varlık içerir. Bir *Varlık* (kişi veya şirket) bir belgenin yazarı veya bir sosyal ağ



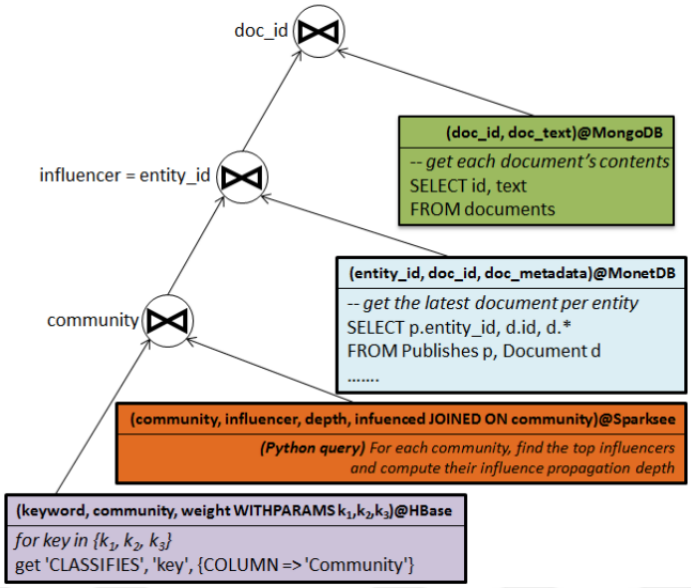
Şekil 4.3: CloudMdsQL sorgu motoru bileşenleri

hesabından bahseden hesaptır. Sosyal ağlarda kopyalar, referanslar ya da ifadeler ile insanların etkileşimleri, etkilerin bir dizisi olarak anlaşılabilir. Başka bir deyişle, kimin neyi ve neyi etkilediğini sorabiliriz. Uygulamaya yeni bir tweet geldiğinde, bu *Etkiler* ve *Topluluklar* aşamalı olarak hesaplanır ve bu nedenle, bu kavramlar uygulama şemasının parçasıdır.

Uygulamanın kullandığı ana sorgunun Q_1'in spesifikasyonu şu şekildedir: k_1,k_2,k_3 anahtar sözcükleri kümesi, en büyük 10 toplulukları bulmak ve her topluluk için en çok etkilenen 20 kişiyi bulmak. Bu etkileyicilerin her biri için sistem, topluluk içindeki etkilenen varlıkların sayısını, etkileyicinin kimliğini, adını ve hesap oluşturma tarihini ve son yayınlanan belgeyi döndürmektedir.

Bu kullanım durumunu uygulamak için, *Etkiler* çizgesini saklamak ve *Toplulukları* hesaplamak için bir çizge veritabanı (Sparksee) kullanılmaktadır; *Varlıklar* ve *Belgeler* (yalnızca meta veriler) ile ilgili tüm temel bilgiler için ilişkisel bir veritabanı (MonetDB); Belge içeriğini saklamak için bir doküman veritabanı (MongoDB); ve anahtar kelime başına toplulukları endekslemek için bir anahtar-değer veri deposu (HBase) kullanılmıştır. Q_1 sorgusunun işletimi Şekil 4.4'te gösterilmiştir.

İkinci kullanım örneği uygulaması, DBLP ve CORDIS bilgi tabanını dikkate alarak belirli bir Avrupa projesi için yorumcuları önerir. DBLP, 1,8 milyon yayın ve 1 milyon yazar içeren bilgisayar bilimi odaklı bir bibliyografik veri kümesidir. CORDIS, 40000 proje ve 1000 kurum içeren Avrupa proje veri setidir. Ana



Şekil 4.4: CloudMdsQL kullanım durumu örneği sorgu işletimi (Kolev et al.'dan 2016b)

sorgu Q_2, araştırmacılara öneriler sunmak için Sparsity-Technologies tarafından oluşturulan bir sistemin temel işlevlerinden biridir.

Bu bilgi sisteminin şeması katılımcıları *Kurum* olan ve bunlardan biri koordinatör olan *Projeleri* içermektedir. Öte yandan, şemanın bir kısmı, her yıl için ilgili kuruluşlara (*Kurumlar*) sahip *Belgeler* (makaleler) ve yazarlarını (*İnsanları*) içeren bir bibliyografik veri kümesini saklar. Bu bilgi sistemi ayrıca*Anahtar Kelimeler* ile *Projeleri* ve *Belgeleri* endeksler; böylece her *Anahtar Kelime* için en iyi uzman *Kurumlar* ve *Kişiler*'in analizi gerçekleştirilebilir.

Uygulama ve Q_2 sorgusu proje üyeleri ile çakışan ilgi alanlarını çözmek için bir çizge veritabanı (Sparksee) kullanmaktadır; çünkü çizge veritabanları ilişkileri çözmek için oldukça verimlidir. Önerilen yorumcularla ilgili alanların tam listesini saklamak ve almak için bir ilişkisel veri deposu (LeanXcale); Anahtar kelimelerle hızlı aramanın avantajından yararlanarak konular listesinde en iyi uzmanları bulmak için bir anahtar-değer veri deposu (HBase); ve önerilen yorumcular tarafından üretilen son kağıdın içeriğini almak için bir belge veri deposu (MongoDB) kullanılmıştır.

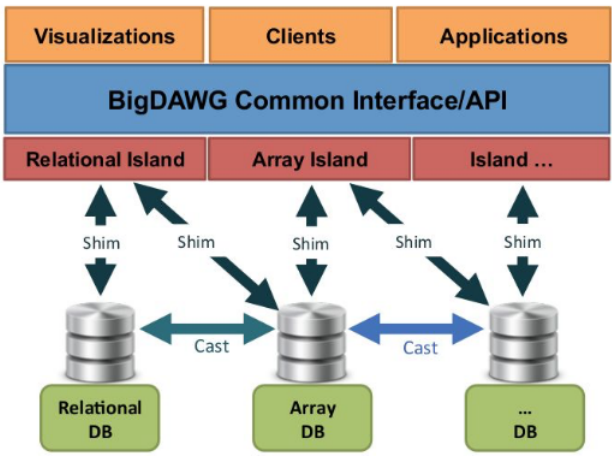
4.2 BigDAWG

Duggan vd. (Duggan et al., 2015) birden fazla veri modelini kapsayan bir bilgi yönetimi ihtiyacına vurgu yaparak yeni bir birleştirilmiş veritabanı (federated database) bakış açısı olan çoklu depo yaklaşımını ortaya atmışlardır. Intel Science

ve Technology Center for Big Data gruplarıyla ortak çalışarak, MIMIC-II (Saeed et al., 2011) hastane veri kümesi üzerinde farklı veri modellerini kapsayan bir kullanım durumu oluşturmuş ve tasarlanan BigDAWG (Duggan et al., 2015; Gadepally et al., 2016; Dziedzic et al., 2015; She et al., 2016) isimli çoklu depo sistemi prototipi bu kullanım durumu üzerinde denenmiştir.

Bu yeni yaklaşımda üç hedef üzerine odaklanılmıştır. Bunlardan ilki saklanacak nesneler için "yer şeffaflığı"dır (location transparency) ki böylece kullanıcılar birden çok veri yönetim sistemini kapsayan bildirimsel sorgular atabilmektedir. Çalışma kapsamında ortaya atılan "bilginin adası" (island of information) soyutlaması tek bir sorgu dili ile erişilen saklama motorlarını ifade etmektedir. İkinci hedef "anlamsal bütünlüktür" (semantic completeness) ve öyle ki yeni bir saklama motoru çoklu depoya eklendiğinde, bu saklama motorlarıyla ilgili herhangi bir işlevsellik kaybedilmeyecektir. Üçüncü hedefte ise kullanıcılar belirli bir arka-uçta (back-end) saklanan nesnelere farklı adalar üzerinden erişebileceklerdir. Örneğin; fiziksel olarak belirli bir arka uçta tutulan bir veriye hem dizi (array) hem de ilişkisel (relational) adalar üzerinden erişilebilecektir.

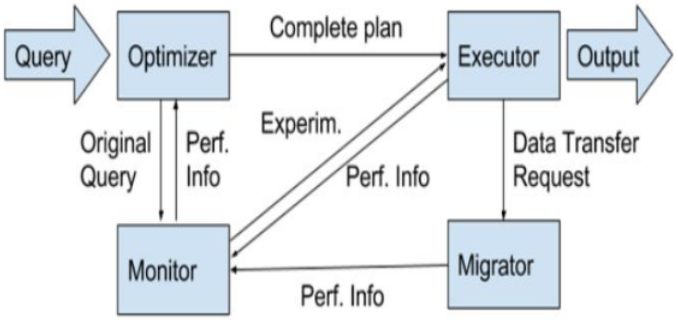
Çoklu depo sisteminin genel mimarisi Şekil 4.5'te görülmektedir. Buna göre sistem veritabanı ve saklama motorları (database and storage engines), adalar (islands), orta katman (middleware) ve API, uygulamalar (applications) olmak üzere 4 katmandan oluşmaktadır. Ana katman BigDAWG katmanı olmakla birlikte yazarlar çalışmada bu 4 katmanı sırasıyla açıklamışlardır.



Şekil 4.5: BigDAWG çoklu depo sistemi mimarisi (Gadepally et al'dan 2016)

İlk katman olan ***veritabanı ve depolama motorları*** katmanında BigDAWG altyapısındaki çoklu depo ortamı bulunmaktadır. Bunun gerekliliği MIMIC-II kullanım durumunda belirtilmiştir.

İkinci sıradaki katman ***adalardır***. Her adanın bir veri modeli, sorgu dili veya operatör kümesi ve bir ya da daha fazla veritabanı motoru bulunmaktadır. Bu sayede adalar kullanıcı için bir soyutluk sunarak, anlamsal bütünlük ve yer şeffaflığı arasında seçim yapılabilmesini sağlar. Kullanıcı scope operatörünü kullanarak alt sorgunun çalışacağı adayı belirtir.



Şekil 4.6: BigDAWG ara katmanı (Gadepally et al'dan 2016)

Üçüncü katman ***BigDAWG*** katmanıdır ve bu katman da kendi içerisinde birkaç bileşenden oluşmaktadır. Bu katmanın ana bileşeni BigDAWG ara-katmanıdır (middleware). Ara katman sorguların karşılanması, sorgu planlama, etkin işletim stratejilerini belirleme, önceki sorguların tarihçesini ve performansını tutma gibi işlerden sorumludur. Şekil 4.6'da yapısı görülen ara katmanın; sorgu planlama (planner) (Gupta et al., 2016), performans izleme (monitor) (Chen et al., 2016), veri taşıyıcı (migrator) (Dziedzic et al., 2016) ve sorgu işletimi (executor) (She et al., 2016) olmak üzere 4 modülü bulunmaktadır. Sisteme gelen sorgu, planlayıcı tarafından ayrıştırılarak bir sorgu planı ağacı kümesi yaratılır. İzleyici bu sorgu planı ağacı kümesini alarak varolan performans verisine göre en uygun ağacı belirler. Çalıştırıcı belirlenen sorgu planı ağacını alarak elindeki verilere göre en uygun yöntemi belirleyerek sorguyu çalıştırır. Ayrıca sorgu çalıştırıcı, taşıyıcıyı kullanarak sorgu motorları ve adalar arasında nesnelerin taşınmasını sağlayabilir. BigDAWG katmanının diğer bileşeni BigDAWG API'dir ve sorguların işletimi için bir arayüz sunar. Sunucu ve istemci bileşenlerinden oluşan API'nin sunucu bileşenleri veritabanlarını birbirine bağlamak için dolgu (shim) denen yapıları kullanır. Dolgular bir adanın yerel sorgu dilinden veritabanının doğal diline dönüşümü sağlamaktadır. *Scope* anahtar kelimesiyle sorgunun çalıştırılacağı ada belirlenirken *Cast* anahtar kelimesiyle de adalar arasında veri dönüşümü gerçekleştirilir. BigDAWG katmanının son bileşeni Çoklu Depo Sorgularıdır. Sorgular genellikle birden fazla adayı kapsadığından etkin bir sorgu işletimi kritik olmaktadır. Sorgular öncelikle eğitim (training) modunda çalıştırılarak mümkün en iyi sorgu planları oluşturulur. Daha sonra üretim (production) aşamasında ise en iyiye yakın olan sorgu planı seçilmeye çalışılarak hızlı çalıştırma amaçlanır. Tasarlanan sistemdeki özgün yanlardan biri de şemalar hakkında bir üst-veriye

sahip olmayı gerektirmeden kara kutu (black box) yaklaşımıyla sorguların işletileceği arka-uçların belirlenmesi ve sorgu eniyileştirmesinin sağlanmasıdır.

Son katman ***uygulama ve görselleştirme*** katmanıdır. Uygulamalar, istemciler ve görselleştirme araçları, API ve ara katman üzerinden çoklu depo sistemini kullanırlar. Ada soyutlaması kullanıcılara kendi uygulamalarını en çok ilgilendiren veri deposunu seçip faydalanmasına olanak tanımaktadır.

Elmore vd. (Elmore et al., 2015) MIMIC II hastane verisi üzerinde oluşturulan kullanım durumu örneğini anlatmışlardır. MIMIC II veri seti içerisinde bulunan veriler şu şekildedir: Dalga formu verileri (başucu cihazlarından 125 Hz'ye kadar ölçümler), hasta meta verileri (isim, yaş, vb.), doktor ve hemşire notları (metin), laboratuar sonuçları ve dolu reçeteler (yarı yapılandırılmış veriler). Pratikte bir hastane, mevcut hastalardan alınan gerçek zamanlı beslemelerle zenginleştirilmiş tüm tarihsel verileri saklayacaktır. Bu nedenle, bu sistem çeşitli veri tiplerini, standart SQL analizlerini (örneğin, belirli bir ilaç kaç hastaya verildi), karmaşık analitiği (örn., Hastanın dalga formu verilerinin FFT'sini hesaplar ve daha sonra "normal" ile karşılaştırır), metin aramalarını (örneğin, belirli bir ilaca veya tedaviye iyi yanıt veren hastaları bul) ve gerçek zamanlı izlemeyi (örn. anormal kalp ritimlerini saptamak) desteklemelidir.

BigDAWG, MIMIC II verisini birden fazla veri deposu içerisinde depolamaktadır. Postgres veritabanı içerisinde hasta meta verileri, tarihsel dalga formu verisi zaman serisi veritabanı olan SciDB içerisinde, aygıt verisi akışı S-Store içerisinde ve ilişkili metin verileri bir anahtar-değer deposu olan Apache Accumula içerisinde depolanmıştır. Veriler üzerindeki etkileşimler şu şekilde sınıflandırılmıştır:

- **Tarama:** Bu, bir kullanıcının tüm MIMIC II veri kümesine göz atabileceği ve daha ayrıntılı bilgilere erişmek için talep üzerine detaylandırma yapabildiği bir pan/yakınlaştırma arabirimidir. Bu arayüz, en üst düzey bir görünümü etkin bir şekilde gösterebilmekte ve kullanıcıların, verileri farklı ayrıntı düzeylerinde araştırmasını mümkün kılmaktadır. Etkileşimli yanıt süreleri sağlamak için, bu bileşen, ScalaR ile kullanıcı hareketlerini tahmin ederek verileri önceden getirmektedir.

- **Keşif Analizi:**Kullanıcılar, kılavuzsuz veri madenciliği kullanarak tıbbi verilerdeki ilginç ilişkileri etkileşimli olarak keşfedebilmektedir. Örneğin seçilmiş hasta popülasyonundaki ırk ve hastanede kalış süresi arasındaki alışılmadık bir ilişki keşfedilir. Bu popülasyon, verilerin geri kalanında

görülen trendi tersine çevirmektedir. Grafikten, kullanıcı, bu korelasyonu neyin tetiklediğini belirlemek için kaynak verilere bakma fırsatına sahiptir.

- **Karmaşık Analizler:** Bu ekran, programlayıcı olmayan bir kişinin, doğrusal regresyon, hızlı fourier dönüşümü ve temel bileşen analizi gibi çeşitli karmaşık analizleri hasta verileri üzerinde çalıştırmasını sağlar.

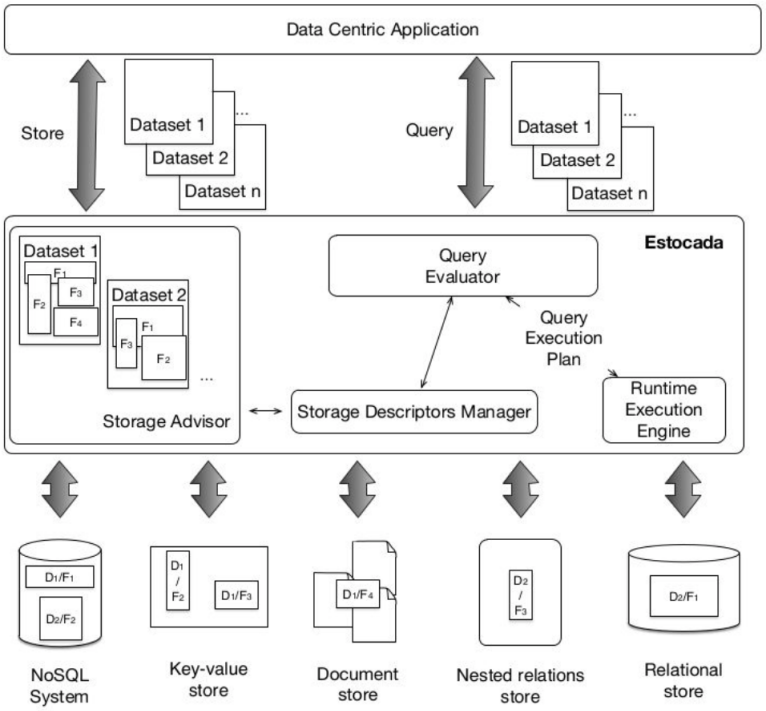
- **Metin Analizi:** Bu pencereden bir kullanıcı, "en az üç doktorun raporunda 'çok hasta' dediği hastaları ve belirli bir ilacı alan hastaları bul" gibi karmaşık anahtar kelime aramalarını çalıştırabilir.

- **Gerçek Zamanlı İzleme:** Mevcut hastaların gerçek zamanlı dalga formu verileri, yeni bir işlem akışı işleme motoru olan S-Store'da saklanmaktadır. Bu ekranda, gerçek zamanlı hasta verileri gösterilmektedir. Ayrıca, gelen dalga formlarını referans olanlarla karşılaştıran bir iş akışı bulunmaktadır ve bu ikisi arasındaki anlamlı farklar belirlendiğinde bir uyarı verilmektedir.

4.3 ESTOCADA

Bugiotti vd. (Bugiotti et al., 2015a) çalışmalarında veri kümesi parçalama (fragmentation) yöntemiyle ölçeklenebilir ve kendi kendini düzenleyebilen (self-tuning) ESTOCADA (Bugiotti et al., 2015a; Bugiotti et al., 2015b; Bugiotti et al., 2016) isimli bir çoklu depo sistemi ortaya koymuşlardır. ESTOCADA her veri kümesine uygulama veya programlama katmanında, doğal yapısında erişme imkanı sunarken, arka uçta ise bu veri kümelerini veya veri kümelerinin parçalarını (fragments) birbirinden farklı depolarda saklamaktadır. Verilen işyükü (workload) bilgisine göre her depoya yüklenecek veri kümesi parçalarını otomatik olarak seçerek performans eniyileştirmesi sağlamaktadır. Sorgu işlemenin temelinde LAV ve görünüm-tabanlı yeniden yazma (view-based rewriting), depo düzenlemenin (storage tuning) temelinde ise görünüm seçme (selection) yatmaktadır. Sistemin mimarisi Şekil 4.7'da görülmektedir.

Sistem bu mimari üzerinden açıklanacak olursa; en üst katmanda veri merkezli uygulama bulunmaktadır. Uygulamalar sisteme saklama ve sorgulama olmak üzere iki tip veri erişim isteği yollamaktadır. Saklama Danışmanı (Storage Advisor) isimli modül veri kümesini, çakışabilmesi (overlapping) mümkün parçalara ayırıp, her bir parçanın saklama görevini de altyapıdaki bir veri yönetim sistemine yönlendirir. Parçalama işlemi veri erişimi isteklerinin işyüküne ve veri modellerine göre belirlenen bir sezgi kümesine göre gerçekleştirilmektedir. S_k deposunda bulunan D_i/F_j veri kümesi parçası için bir $sd(S_k, D_i/F_j)$ saklama



Şekil 4.7: ESTOCADA mimarisi (Bugiotti et al.'dan 2015)

tanımlayıcısı yaratılır ve böylece hangi depoda hangi veri kümesinin hangi parçasının bulunduğu öğrenilmiş olur.

Saklama Tanımlayıcısı Yöneticisi (Storage Descriptor Manager), varolan saklama tanımlayıcılarının kümesinin kataloğunu tutar. Ayrıca SDM her bir sd saklama tanımlayıcısı için C_{sd} maliyet bilgisini de tutar ve böylece her D_i/F_j veri kümesi parçası için veri erişim maliyetleri hesaplanmış olur. SDM ayrıca her saklama tanımlayıcısının kullanım sıklığını da tutar ve Saklama Danışmanı bu bilgiyi kullanarak parçalar üzerinde ekleme/çıkarma/güncelleme işlemleri yapabilir.

Sorgu Değerlendirici, bir D_i veri kümesine uygulamalar tarafından yollanan orijinal sorguyu alır ve bu sorguya uygun şekilde cevap verebilmek için ilgili D_i veri kümesinin bulunduğu parçaları saklama tanımlayıcıları üzerinden bulur. Bu parçalar (materialized views) üzerinden alternatifleri de hesaba katarak sorguyu yeniden yazar. Yeniden yazdığı bu sorgular arasından en iyi performansa sahip olacağını düşündüğü sorguyu çalışma zamanı bileşenine iletir. Çalışma Zamanı İşletim Motoru (Runtime Execution Engine), mantıksal planı fiziksel plana çevirerek sorgu parçalarının doğrudan depolar üzerinde ve birleştirme operasyonlarını gerçekleştiren ESTOCADA'nın kendi çalışma zamanı motoru üzerinde işletilmesini sağlar. Fiziksel planın işletilmesi sonucu sonuçlar istemci uygulamaya döndürülür.

Bugiotti vd. (Bugiotti et al., 2016) yaklaşımlarını geniş ölçekli çevrimiçi satış (pazar yeri) uygulaması üzerinde sınamışlardır. Pazar yeri, kullanıcı tarafından üretilen verileri hem aktif (siparişler, ürün incelemeleri vs.) hem de pasif olarak (kaydedilen veb günlüklerine göz atma) kullanarak müşteri deneyimini geliştirirken satışları en üst düzeye çıkarmayı hedeflemektedir.

Pazar yeri uygulamasının veri modeli şu şekildedir: Ürün kataloğu JSON dokümanları olarak düzenlenmiştir; Kullanıcı verileri (koordinatlar, ödeme bilgileri, vb.), sipariş ve gönderim bilgileri bir dizi ilişki içerisindedir, alışveriş kartları dokümanlardır, veriler ise kullanıcının pazardaki etkileşimini kaydeden veri HTTP günlük dosyalarındadır. Ürün katalogları SOLR'da (tam metin indeksleme ve Lucene bazında arama olanağı) saklanır, kullanıcı hesapları, tercihler, siparişler ve gönderiler bir Postgres kümesi içinde depolanmaktadır. MongoDB bir yandan kredi kartı bilgilerini depolamak için kullanılırken diğer yandan bir kümesinde de web günlüklerini depolamak için kullanılır. Apache Spark kullanıcıların MongoDB üzerinde tutulan web sitesi ziyaretleri hakkındaki bilgileri ve istatistikleri işlemek için kullanılır.

Bugiotti vd. (Bugiotti et al., 2016) bu veri modeli şemasının bir süre kullanıldıktan sonra değiştirildiğini belirtmektedirler. Senaryonun işletilmesi sırasında baskın sorguların (bir tarafta kullanıcı tercihleri ve diğer alışveriş kartları) anahtar temelli aramalara karşılık geldiği fark edilmiştir. Bu yüzden geliştirme ekibi ilgili veri parçalarını depolamak için Voldemort anahtar-değer deposunun kullanımını araştırmaya karar vermiştir.

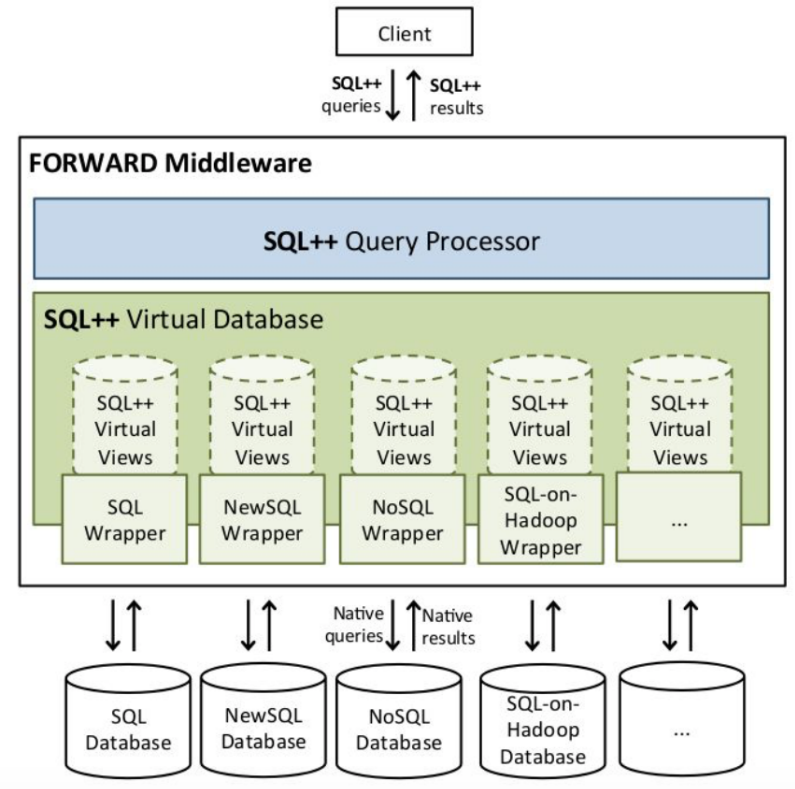
Ortaya çıkan performansı ölçmek ve hangi deponun kullanılacağına karar vermek için bu parçaları Voldemort'a taşımanın (verilerin farklı bir veri modelinde yeniden yapılandırılması gerektiği gibi hatalara açık bir süreç); uygulamayı daha önce kullanılan belge ve ilişkisel depolar yerine anahtar-değer deposuyla etkileşime uyarlamanın gerekli olduğu vurgulanmıştır. Sonuç olarak bu değişiklik, uygulama işyükünde %20'lik bir performans artışı getirmiştir.

Diğer bir değişilik de kişiselleştirilmiş ürün arama sorgularının yarattığı darboğaz ve ekstra bakım yüzünden yapılmıştır. Bu sorgu, bir kullanıcı aramasına yanıt olarak ilk önce gösterilmesi gereken ürünleri tanımlamak için kullanıcı geçmiş satın alımları (ilişkisel mağazadan) ve tarama geçmişi günlüğünü birleştirmektedir. Bu sorguyu hızlandırmak için, geçmiş satın alımlar ile geçmiş arama verileri arasında birleştirme sonucunun Spark içinde depolanmış iç içe bir ilişkide gerçekleştirilmesi önerilmiştir. Ayrıca bu ilişki kullanıcı kimliği ve ürün kategorisi tarafından endekslenmelidir. Bu değişiklik, performansı % 40 oranında arttırmıştır.

Ancak iş ihtiyaçları yeni sorgulama gereksinimleri getirmiştir ve bu sorguları çalıştırabilmek için daha önce yapılan seçimlerle çatışma yaşanması gereklidir. Yazılım geliştirme ekibi veriyi yeniden taşımak ve uygulamayı yeniden düzenlemek seçeneği ile karşı karşıya olduğu ve insan gücü eksikliğinden dolayı bu işlemi gerçekleştirememiştir.

4.4 FORWARD

Ong vd. (Ong et al., 2014b) çalışmalarında birbirinden farklı modellere ve dil yapılarına sahip NoSQL, NewSQL ve SQL-on-Hadoop veritabanlarını SQL++ (Ong et al., 2014b; Ong et al., 2014a; Ong et al., 2014c; Papakonstantinou, 2016) dilinin birleştirici yönlerini kullanarak birbiriyle kıyaslamışlardır. Çalışmalar farklı yapıdaki veritabanlarının yetenek bakımından birbiriyle kıyaslanması için bir inceleme (survey) niteliğinde olup bu kıyaslamanın yapılabilmesi için veritabanlarını bütünleştiren bir yapı gereklidir. Yazarlar bu bütünleştirme için gerçekleştirmiş oldukları FORWARD (Ong et al., 2014b; Ong et al., 2014a; Ong et al., 2014c; Papakonstantinou, 2016) isimli ara-katman sorgu işlemcisini kullanmışlardır. SQL++ da FORWARD tarafından kullanılmakta ve birleştirici bir veri modeli ve yarı-yapısal sorgu dili olarak işlev görmektedir. Şekil 4.8'de FORWARD sorgu işlemcisinin yapısı görülmektedir.

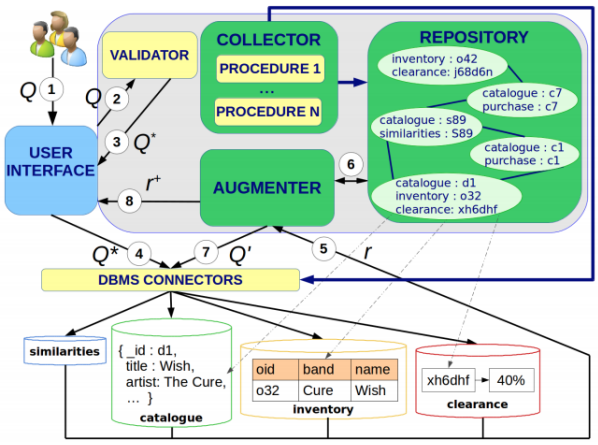


Şekil 4.8: SQL++ tabanlı FORWARD sanal veritabanı sorgu işlemcisi (Ong et al.'dan 2014)

Kavramsal olarak her veritabanı istemciye SQL++ kaynağı olarak görülmektedir. İstemci bir veya birden fazla veri kaynağı üzerine SQL++ sorgusu yollamakta ve FORWARD sorgu işlemcisi de bu sorgunun çalışması için bir ya da birden fazla veritabanının yerel sorgu diline dönüşümünü gerçekleştirmektedir. FORWARD sorgu sonuçlarını da veritabanları ile SQL++ arasındaki uyumsuzlukları gidererek gerektiğinde bir ya da daha fazla veri modelini kapsayacak şekilde bütünleşik olarak birleştirmektedir. Yazarlar FORWARD ara-katman sorgu işlemcisinin detaylarını yayınlayan bir çalışma yapmamışlardır. Bunun yerine bir kullanım durumu üzerinden işlemcinin sorguyu işleyiş biçimi açıklanmıştır (Ong et al., 2014c).

4.5 QUEPA

Maccioni vd. (Maccioni et al., 2016) ortaya koydukları QUEPA isimli araç ile bir çoklu depo ortamında artırma (augmentation) yöntemiyle sorgulama kolaylığı sağlamaktadırlar. Bu araç herhangi bir ortak veri modeli ve konfigürasyon ihtiyacı doğurmadan, doğrudan tak-çalıştır modda kolaylıkla bütünleşip çalışabilmektedir. İlk sorgulamadan sonra kullanıcı olası sorgu kombinasyonlarına (guided-search) yönlendirilmektedir.



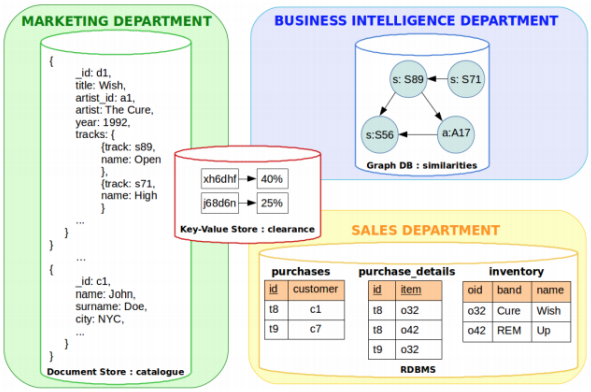
Şekil 4.9: QUEPA mimarisi (Maccioni et al.'dan 2016)

Şekil 4.9'da gösterildiği gibi, QUEPA'nın mantıksal mimarisi üç ana bileşene dayanmaktadır:

- Yerel sorguların sonucunu arttıran bir arttırıcı
- Farklı veri depolarına ait veri nesneleri arasında anlamlı ilişkilerin kaydını tutan bir bağlantı deposu

- Bu tür ilişkileri toplayan bir bağlantı toplayıcısı.

QUEPA aracı ile çevrimiçi müzik satan ACME şirketi için Last.fm veriseti üzerinde Şekil 4.10'da görüldüğü gibi bir çoklu depo ortamı oluşturulmuştur. Geliştirilen senaryolarda satış departmanı, ACID işlemlerine ihtiyaç duymakta ve satıcılar tarafından sağlanan satın almalar ve ürünlerin envanteri için ilişkisel veritabanlarını kullanmaktadır. Pazarlama departmanı, her bir öğenin bir JSON belgesiyle temsil edildiği bir müzik ve müşteri kataloğu için belge deposunu kullanmaktadır. İş zekası departmanı, öğeler arasındaki benzerlikleri temsil etmek için çizge veritabanlarını kullanmaktadır. Buna ek olarak, yukarıda anlatılan üç departman arasında paylaşılan ve halihazırda gümrükte bulunan ürünleri içeren bir anahtar-değer deposu bulunmaktadır.



Şekil 4.10: QUEPA çoklu depo ortamı (Maccioni et al.'dan 2016)

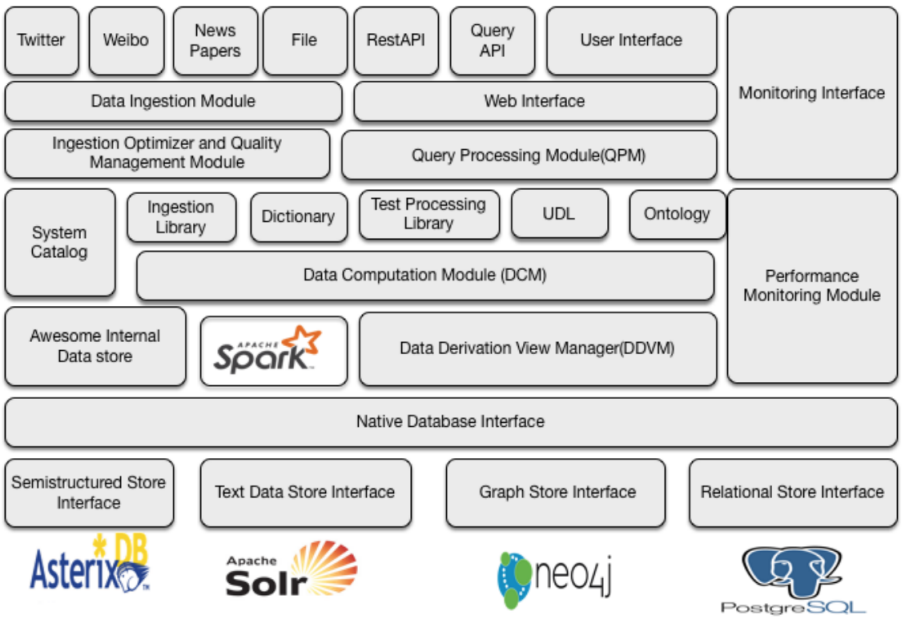
4.6 AWESOME

Dasgupta vd. (Dasgupta et al., 2016) sosyal veri çözümlemesini de desteklemek için AWESOME isimli bir çoklu depo sistemi geliştirmişlerdir. AWESOME sistemi çözümlemeli uygulamalarda kullanılan türetilmiş veriyi işleyebilmek için tasarlanmıştır. ADIL adında tasarlanılan veri alınım dili (a data ingestion language) ile ham ve türetilmiş veri üzerinde taşıma ve çözümleme yapılabilmektedir.

AWESOME sisteminin altında yatan çoklu depo, birçok modern büyük veri uygulamasının iki tür veri heterojenliği ile karşılaştığı gözlemine dayanmaktadır. İlk kategori, uygulamanın karmaşık olduğu ve belirli bir veri kümesinin belirli bir veri modeline ait olduğu heterojen veri kümelerini depolaması, sorgulaması ve analiz etmesi gerektiğinde ortaya çıkar, ancak analizin bunları rasgele karmaşık yollarla birleştirmesi gerekir. Örneğin, bir müşteri analizi uygulaması, müşteri

verilerinin yanı sıra sosyal bir forumdan müşteri geri bildirimleri şeklinde sosyal verilere ihtiyaç duyar. Buna uygulama düzeyinde büyük çeşitlilik problemi denmektedir. İkinci uygulama kategorisi yine de birden fazla kaynaktan gelen verilerle ilgilenebilir, bu durumda, verilerin kendisi, veri kümesinin bir bölümünün ilişkisel bir model için en uygun olduğu, diğer bir bölümünün de çizge verileri olarak daha uygun olabileceği, kendiliğinden karıştırılmış bir modeldir. Örneğin, mikrobloglar için meta veriler düz ilişkilerde temsil edilebilir, ancak yanıt-cevap, takip edici linkler, çizge odaklı bir analiz için daha uygun olabilir.

AWESOME sistemi, bir çoklu depo sistemi kullanarak her iki çok çeşitliliği de ele almak üzere tasarlanmıştır. Çoklu depo, dört tip veri deposu ve bir hesaplama motoru üzerine kurulmuştur - her veri deposu yerel olarak belirli bir veri modelini tanır ve motor bir dizi hesaplama yapısını kabul edebilir veya yayabilir. Veri depoları, metin için kullanılan ilişkisel, yarı yapılandırılmış, özellik çizgeleri ve vektör modelini destekleyecek şekilde tasarlanırken, hesaplama deposu, BlockMatrix gibi veri yapılarına sahip Spark'dır.



Şekil 4.11: AWESOME çoklu depo sisteminin mimari bileşenleri (Dasgupta et al.'dan 2016)

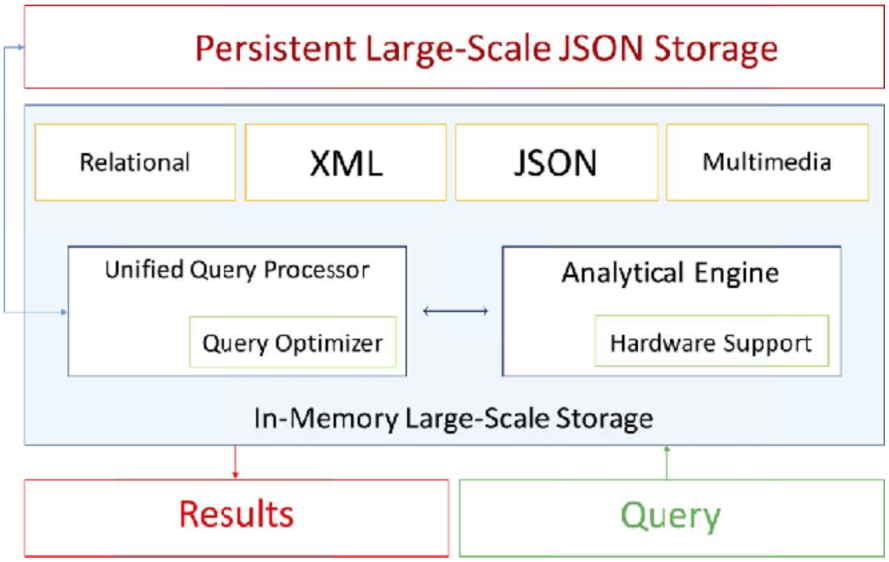
Şekil 4.11'da görüldüğü gibi AWESOME çoklu depo sisteminin bileşenleri, Veri Yutma Modülü (Data Ingestion Module-DIM), AWESOME İç Veri Deposu (AWESOME Internal Data Store-ASTORE), Veri Türetme Görünümü Yöneticisi (Data Derivation View Manager-DDVM), Sistem Kataloğu (System Catalog-SC), Sorgu İşleme Modülü (Query Processing Module-QPM), Veri Hesaplama Modülü (Data Computation Module-DCM), Analitik Kitaplıkları (Analytics Libraries-AL) ve Performans İzleme Modülü'nden (Performance Monitoring Module-PMM) oluşmaktadır. DIM, dosya kaynaklarından, harici veri hizmetleri tarafından

sağlanan veri API'lerinden ve veri akışı API'lerinden veri kabul etme görevini yürütmektedir. ASTORE, AWESOME'a dahil olan ve çoklu depo sistemindeki standart depolardan herhangi birinde saklanamayan veri nesnelerini saklar. DDVM, veri türetme sürecini, gerektiğinde model dönüşümü ile birlikte bir veritabanı görünümü sistemi üzerinden yönetir. SC, tüm veri kaynaklarının ve türetilmiş görünümlerin kaydını tutar. QPM, çoklu depo sistemine yapılan sorguları ayrıştırmak, planlamak, optimize etmek ve yürütmekle sorumludur. DCM, sistemdeki tüm hesaplama süreçlerini yönetir. AL, kullanıcılar tarafından veya AWESOME'ın tanıdığı veri türlerini kullanarak verileri birbiriyle değiştirecek şekilde sarılmış üçüncü taraf kütüphaneleri tarafından sağlanan algoritmaları içerir. PMM, her depoda ve AWESOME sisteminin geri kalanında kullanılan kaynakları (bellek, IO işlemleri vb.) izler ve bu bilgiyi sadece sistem davranışını izlemek için değil, aynı zamanda sorgu işleme modüllerini planlamak için de kullanır.

4.7 HYBRID.POLY

Podkorytov vd. (Podkorytov et al., 2017) heterojen büyük ölçekli veri setlerinin depolanması, erişilmesi ve analiz edilmesi için ortak bir zemin oluşturacak olan bir analitik çoklu depo sistemi olan HYBRID.POLY'i tanıtmışlardır. HYBRID.POLY mimarisi Şekil 4.12'de gösterilmiştir. HYBRID.POLY'nin bellek içi depolama altyapısı, çeşitli veri modellerini desteklemektedir. Sorgulama arabirimi, kullanıcı sorgularını, ilişkisel verilerle birlikte ilişkisel olmayan veriler üzerinde karmaşık analitik sorgulamalar yapmak için ekstra yetenekler sağlayan SQL'in bir üst kümesi olan karma bir dilde kabul etmektedir. Sorgu, binlerce düğüm içeren büyük melez sorgu planlarını işleyebilen bir hibrid optimize edici tarafından otomatik olarak optimize edilmektedir.

HYBRID.POLY sistem mimarisi Sorgu Dili, Sorgu İşleme Motoru, Depolama Motoru ve Analatik Sorgu İşleme İçin Donanım Hızlandırma isimli dört ana bileşenden oluşmaktadır. Sorgu dili için iki farklı yaklaşım mevcuttur. Kullanıcıların interaktif sorguları çalıştırabilmesi için Apache Spark ve Pig'de desteklendiği gibi bir betik dili kullanılabilmektedir. Diğer yaklaşım ise SQL üzerinde tasarlanmış bir sorgulama dilidir. Sorgu işleme altyapısı, sorgu ayrıştırıcısından, sorgu derleyicisinden ve sorgu iyileştiriciden oluşur. Sorgu ayrıştırıcısı, kullanıcının sorgusunu işler ve bir Özet Sözdizimi Ağacı (AST) derler. Sorgu derleyicisi, AST'yi depo üzerinde çalıştırabilmek için iç sorgu temsiline derler. İki tip sorgu iyileştiricisine sahiptir. Statik sorgu iyileştirici, daha ucuz bir yürütme planı elde etmek için AST'yi yeniden yazar. Dinamik sorgu iyileştirici ise en uygun yürütme planını seçmek için çalışma zamanındaki veriler hakkındaki



Şekil 4.12: HYBRID.POLY mimarisi (Podkorytov et al.'dan 2017)

bilgileri kullanır. HYBRID.POLY'yi isteğe bağlı kalıcılık içeren dağıtılmış bir çok modelli depolama motoru kullanmaktadır. Dağıtılmış depolama daha büyük hacimli verileri ele almayı sağlar. Bellek içi depolama, gecikmeyi azaltır, karmaşık analitik sorgular için etkileşimli performans sağlar. Kalıcılık, hata toleransı için gereklidir. Son olarak, heterojen verilerin depolanması için farklı modelleri desteklemektedir. Son olarak HYBRID.POLY analitik sorgu işlemini hızlandırmak için, bir dizi yeni NVIDIA GPU'yu veya Google TPU'ya benzer özel bir matris işleme birimini desteklemektedir.

Desteklenen veri modelleri; İlişkisel veri modeli, dizi veri modeli, XML veri modeli, JSON veri modeli, çizge veri modeli olarak belirtilmiştir. Ayrıca resimler, ses dosyaları, lokasyon bilgileri için farklı veri destekleri sunmaktadır.

4.8 Çoklu Veri Depolarının Sınıflandırılması

Bu bölümde yukarıda anlatılan çoklu depolama sistemlerinin sınıflandırılması sağlanmış ve yukarıda değinilmeyen çoklu veri depolama sistemleri bu başlık altında incelenmiştir.

Daha önce çoklu depo sistemlerini sorgulama yöntemleri açısından yerel depolara bağımlılıklarına göre "gevşek-bağımlı", "sıkı-bağımlı" ve "hibrit" olarak sınıflandırıldığından bahsedilmişti (Bkz. Bölüm 3.2). Bondiombouy and Valduriez (2016) sunmuş oldukları araştırma raporunda bu üç farklı sınıfa uyan üçer çalışma tanıtmışlardır. Bu çalışmalardan gevşek-bağımlı olarak; *BigIntegrator* (Zhu and Risch, 2011), *Forward* (Ong et al., 2014b) ve *QoX* (Simitsis et al., 2012);

	Çoklu depo sistemi	Hedef	Veri Modeli	Sorgu Dili	Veri Depoları
Gevşek-bağımlı	BigIntegrator	İlişkisel ve bulut verilerini sorgulama	İlişkisel	SQL-benzeri	BigTable, RDBMS
	Forward	İlişkisel ve NoSQL verilerini birleştirme	JSON-tabanlı	SQL++	RDBMS, NoSQL
	QoX	Çözümleyici veri akışları	Çizge	XML-tabanlı	RDBMS, MapReduce, ETL
Sıkı-bağımlı	Polybase	RDBMS'ten Hadoop'u sorgulama	İlişkisel	SQL	HDFS, RDBMS
	HadoopDB	Hadoop'tan RDBMS'i sorgulama	İlişkisel	SQL-benzeri (HiveQL)	HDFS, RDBMS
	Estocada	Öz-ayarlamalı (self- tuning)	Ortak veri modeli yok	Yerel sorgu dilleri	RDBMS, NoSQL
Hibrit	SparkSQL	SPARK'ın üzerinde SQL	İç içe (nested)	SQL-benzeri	HDFS, RDBMS
	BigDAWG	İlişkisel ve NoSQL'i birleştirme	Ortak veri modeli yok	CAST ve SCOPE operatörleriyle birlikte ada (island) sorgu dilleri	RDBMS, NoSQL, Array DBMS, DSMS
	CloudMdsQL	İlişkisel ve NoSQL'i sorgulama	JSON-tabanlı	Yerel alt sorgularla beraber SQL-benzeri	RDBMS, NoSQL, HDFS

Çizelge 4.1: Çoklu depo sistemlerinin işlevselliği (Bondiombouy and Valduriez'den 2016)

sıkı-bağımlı olarak; *Polybase* (DeWitt et al., 2013), *HadoopDB* (Abouzeid et al., 2009), *Estocada* (Bugiotti et al., 2015a); hibrit olarak ise; *SparkSQL* (Armbrust et al., 2015), *BigDAWG* (Duggan et al., 2015), *CloudMdsQL* (Kolev et al., 2016d) sistemleri örnek olarak seçilmiştir. Anlatılan çalışmaların mimarisi, sorgu işletimi ve eniyileştirme yöntemleri açıklanarak, bu çalışmalar Çizelge 4.1'de (araştırma raporunda Table-1) işlevsellik ve Çizelge 4.2'te (araştırma raporunda Table-2) gerçekleştirim teknikleri olmak üzere farklı bakış açılarından sınıflandırılmış ve değerlendirilmiştir.

	Çoklu depo sistemi	Özel Birimler	Şemalar	Sorgu İşleme	Sorgu Eniyileştirme
Gevşek-bağımlı	BigIntegrator	Alımcı, soğurucu, sonlandırıcı	LAV	Erişim Filtreleri	Sezgisel
	Forward	Sorgu işlemcisi	GAV	Veri deposu yetenekleri	maliyet-tabanlı
	QoX	Veriakışı motoru	Yok	Veri/fonksiyon taşıma, işlem ayrıştırımı	maliyet-tabanlı
Sıkı-bağımlı	Polybase	HDFS köprüsü	GAV	Sorgu ayırma	maliyet-tabanlı
	HadoopDB	SMS planlayıcı, db bağlayıcı	GAV	Sorgu ayırma	sezgisel
	Estocada	Saklama danışmanı	Somutlaştırılmış görünüm	Görünüm tabanlı sorgu yeniden yazımı	maliyet-tabanlı
Hibrit	SparkSQL	Katalizör genişletilebilir eniyileştirici	Veri çerçeveleri	Sütun depolaması ile İç- bellek önbellekleme	maliyet-tabanlı
	BigDAWG	Ada sorgu işlemcileri	Adaların bünyesinde GAV	Fonksiyon/veri taşıma	sezgisel
	CloudMdsQL	Sorgu planlayıcı	Yok	Bağlı birleşim	maliyet-tabanlı

Çizelge 4.2: Çoklu depo sistemlerinin gerçekleştirim teknikleri (Bondiombouy and Valduriez'den 2016)

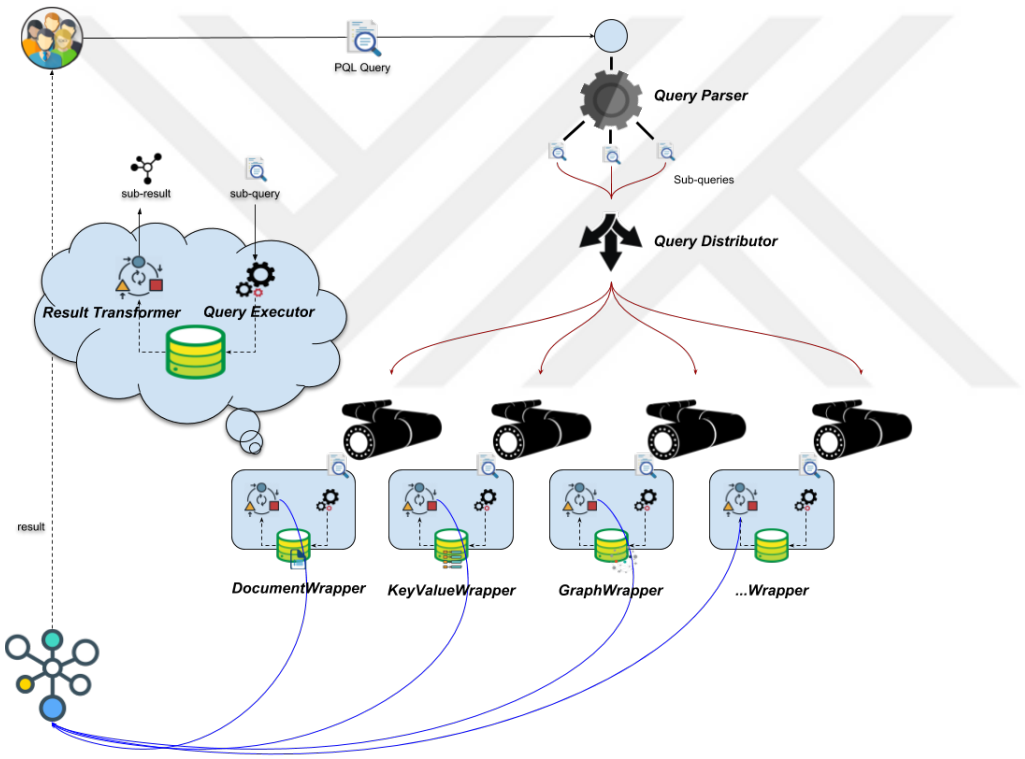
5 ÇOKLU DEPO SİSTEMİNİN GELİŞTİRİMİ

Tezin bu bölümünde geliştirilmiş olan prototip çoklu depo sisteminin mimari yapısı ve bu mimari yapı içerisinde bulunan birimler anlatılacaktır. Çoklu depo sistemleri büyük verinin hız, çeşitlilik ve hacim gereksinimlerine çözüm getirmeyi amaçlamaktadır. Verinin çeşitliliği probleminin, çoklu veri depolarının altında yatan, farklı veri tipleri ve yapıları için özelleşmiş birden fazla NoSQL veri deposunu organize bir biçimde yönetmeyi hedefleyen arabulucu-sarmalayıcı mimari aracılığıyla çözülmesi hedeflenmiştir. Çoklu depo sistemine yeni bir modül eklendiğinde, sistemin kolayca genişleyebilir, esnek bir yapıya sahip olması çeşitliliği yönetmek için kritik önem taşımaktadır. Veriye olan bağımlılık da düşük olmalı ve böylece sistem içerdiği veri açısından farklı bir kullanım durumuna kolayca adapte olabilmelidir. Geliştirilen soyut mimari ile sarmalayıcılar kolayca sisteme entegre edilebilmektedir. Hız ve hacim problemleri de NoSQL veri depolarının iç dinamikleriyle çözülebilmektedir. Ancak prototip tasarımda NoSQL veritabanlarının tek bir makine üzerinde çalıştığı düşünülmektedir. Dolayısıyla hacim problemi, ileriki aşamalarda dağıtık çalışabilirliğin çoklu depo sistemine entegre edilmesiyle çözülebilecektir.

Geliştirilmiş olan prototip çoklu depo sisteminin hedefini farklı veri depoları içerisinde bulunan verinin ortak bir arayüz aracılığıyla sorgulanması oluşturmaktadır. Bunu yaparken ortak bir veri modeli kullanılmamıştır; arabulucu-sarmalayıcı mimari aracılığıyla var olan veri depolarına, veri depoları Java API'leri aracılığıyla erişilmekte ve sorgulama yapılmaktadır. Ancak sorgulama sonucu dönen sonuçlar anahtar-değer veri modeli benzeri bir bellek yapısında tutulmaktadır ve sonuç gösterme işlemi bu veri yapısı aracılığıyla gerçekleştirilmektedir. Sistem veri depolarını sorgulamak için ortak bir sorgu diline sahiptir. Sorgu dili veri depolarına atılacak - veri deposunun kendi diliyle - sorgulara sıkı-bağımlıdır. Sorgu çalıştırma ve veri bütünleştirme işleminde GAV yaklaşımı benimsenmiştir: Sorgu çalıştırma ve sonuç dönüştürme sarmalayıcıların sorumlulukları altındadır. Arabulucu katmanındaki sorgu ayrıştırıcı alt birimi verilen sorgu içerisindeki şema bağlantılarını yakalayarak ilgili sarmalayıcılara iletir ve sonuç dönüştürücü bu bağlantıları kullanarak dönüştürme işlemini gerçekleştirir. Tezin buradaki özgün yapısı ise çoklu depo sistemini sorgulamak için basit bir sorgu dili geliştirimi ve bu sorguların çalıştırılacağı ve sonuç dönüşümlerinin yapıldığı veri depoları sarmalayıcılarının gerçekleştirimidir. Bundan sonraki bölümlerde sistemin mimari yapısı ve modüllerin gerçekleştirimi anlatılacaktır.

5.1 Çoklu Depo Sisteminin Mimari Yapısı

Geliştirilmiş olan mimari Bondiombouy ve Valduriez'in (2016) araştırma raporunda belirttikleri kategorilerden gevşek-bağımlı sistemlere dahil edilebilmektedir. Bu sınıflandırma çoklu depo sistemlerinin altında yatan veri depolarına bağımlılık seviyesine göre yapılmıştır. Ancak sürece sorgu işletimi açısından bakılacak olursa, geliştirilmiş olan çoklu depo sisteminin sıkı-bağımlı olduğu söylenebilir. Çünkü ortak sorgu dili içerisinde verilmiş olan yerel depo sorguları, depo üzerinde doğrudan çalıştırılmakta ve sorgu eniyileştirme, yeniden yazma vb. işlemler deponun kendi sorgu motoru tarafından yapılmaktadır.



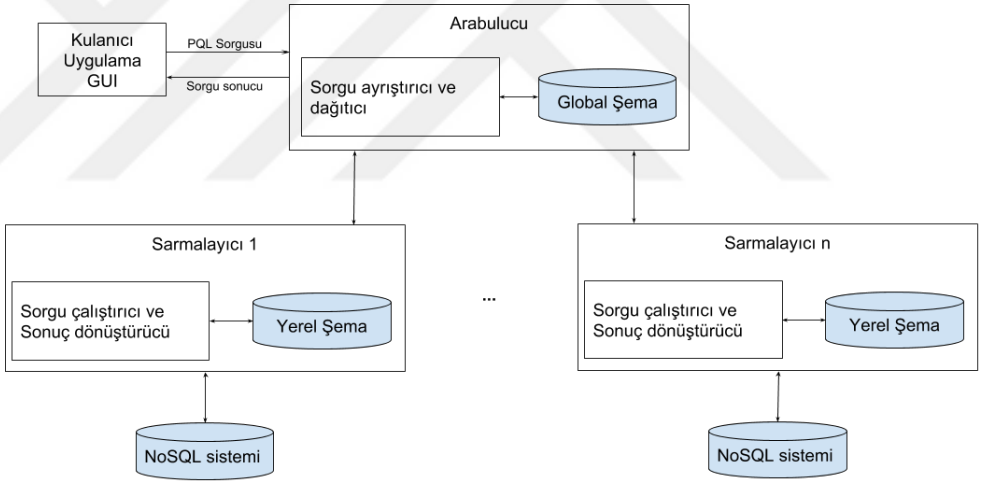
Şekil 5.1: Prototip çoklu depo sistemi mimarisi

Mimarinin gösterimi Şekil 5.1'de yapılmıştır. Buna göre *QueryParser (Sorgu Ayrıştıcı)* sisteme ait sorgu dili içerisinde bulunan ve yerel depolara atılacak olan alt sorguları ayrıştırmakla sorumludur. Elde edilen alt sorgular ilgili sarmalayıcılara *QueryDistributor (Sorgu Dağıtıcı)* aracılığıyla gönderilir ve *QueryExecutor (Sorgu Çalıştırıcı)* sarmalayıcı içerisinde bulunan veri deposu üzerinde, veri deposu arayüzü aracılığıyla sorguları çalıştırmaktadır. *ResultTransformer (Sonuç Dönüştürücü)* ise sorgu sonucu dönen veriyi yerel şemaya eşlemekte ve dönüştürülen veriyi bellek üzerinde işlem gören bir anahtar-değer veri modeli üzerinde saklamaktadır. Tüm alt sorgu sonuçları aynı anahtar-değer veri modeli

üzerinde depolanmakta ve sorgu sonucu olarak birleştirilmektedir.

5.2 Arabulucu-sarmalayıcı Mimarinin Gerçekleştirimi

Bölüm 3'te de anlatıldığı üzere arabulucu-sarmalayıcı mimari veri kaynağı dağılımı, heterojenliği ve özerkliği ile ilgilenmektedir. Prototip çoklu depo sisteminin arabulucu-sarmalayıcı mimarisi de bu özellikleri sağlayabilecek yapıda olmalıdır. Mimari yapıda her veri deposuna ait bir sarmalayıcı bulunmakta ve uygulama içerisinden bu veri depolarına erişim API'ler aracılığıyla gerçekleştirilmektedir. Ancak veri depolarına uygulama dışından da erişim mümkündür: Veri depoları çoklu depo sistemi dışından da kontrol edilebilmekte ve yönetilebilmektedirler. Böylece veri kaynağı özerkliği sağlanmış olmaktadır. Geliştirilen çoklu depo sistemi birbirinden farklı veri modellerine sahip veritabanlarını desteklemekte ve bu veritabanlarının ortak bir sorgu dili aracılığıyla sorgulanabilmesini sağlamaktadır.



Şekil 5.2: Arabulucu-sarmalayıcı mimari

Çoklu depo sisteminin arabulucu katmanında *Sorgu Ayrıştırıcı* ve *Sorgu Dağıtıcı* bulunmaktadır. Sarmalayıcı katmanında ise *Sorgu Çalıştırıcı* ve *Sonuç Dönüştürücü* yer almaktadır. Arabulucu, ilgili depo sorgusunu dağıtmakla yükümlüdür. Dağıtma işlemi aynı makine içerisinde yer alan farklı depolara olabileceği gibi, farklı makinelerde yer alan veri depolarına da olabilmektedir. Prototip çoklu depo sistemi şu anki geliştirim ihtiyaçlarına göre aynı makine üzerindeki depolara erişim sağlamaktadır. Ancak gerçekleştirilen arabulucu-sarmalayıcı mimari dağıtık çalışmaya uygun bir alt-yapı göz önüne alınarak tasarlanmıştır. Sistemin arabulucu-sarmalayıcı mimarisi Şekil 5.2'de görülebilmektedir. Mimari tek bir arabulucu katmanı ile sistemde var olan $n$tane sarmalayıcı üzerinde sorgu çalıştırma işlemininin tetiklenmesini

sağlayabilmektedir. Böylece performans ihtiyacı durumlarında sarmalayıcılar yatay olarak ölçeklenebilecek bir yapıya sahip olmaktadır.

5.2.1 Arabulucu katmanın geliştirimi

5.2.1.1 Çoklu depo sistemi için hafif bir sorgulama dili: PQL

Sorgu Ayrıştırıcı alt-biriminin çalışabilmesi için öncelikle sistemin sıkı-bağımlı çoklu sorgu yapısı (Şekil 5.3) ve içerisinde yer alan operatörler belirlenmiştir. Dilin yapısına göre alt-sorgular ve bu sorguların çalıştırılacakları veri modeli tanımları ayrı ayrı yapılmalıdır. Alt-sorgular **'FROM'** operatörünün içerisinde <*sub*><*/sub*> etiketleri arasında tanımlanmalıdır (Örneğin: <*sub*>$Query_1$</sub>) ve hemen ardından sorgunun çalıştırılacağı veri modeli tanımı yapılmalıdır. Bu veri modeli tanımı **'#model'** anahtar kelimesi ile belirlenmiştir. Böylece *Sorgu Dağıtıcı* alt-birimi sorgunun hangi veri modeli üzerinde çalıştırılması gerektiğini tespit edebilecek ve sorgu çalıştırma işlemini başlatabilecektir. **_{Sorgu}#model** tanımlaması **'as'** operatörüyle haritalanmalıdır, böylece veri modeli içerisinde kayıtlı değerler alınabilmektedir. **SELECT** operatöründen sonra bu tanımlamalar yapılabilmekte ve hangi veri modeli içerisinden hangi alanın sorgu sonucu içerisinde yer alması isteniyorsa bu bölümde tanımlanabilmektedir. Açıklayıcı örneklere Bölüm 6'dan ulaşılabilmektedir. Sorgu dilinin bu tez kapsamında kullanılmayan ancak ileride yapılacak birleştirme işlemlerini gerçekleştirmek için tanımlanmış olan operatörü **'JOIN'** ise veri modelleri içerisinde bulunan değerler içerisinde eşlenik olan özelliklerin tanımlanmasını sağlamaktadır. Eşlenik olan özellikler **'='** (eşitlik) simgesiyle belirlenmiştir.

Sorgu yapısı ve operatörleri belirlendikten sonra bu sorguyu ayrıştıracak bir çözümleyici gerçekleştirimi yapılmıştır. Bunun için açık kaynaklı bir ayrıştırıcı üretici aracı olan ANTLR[11] kütüphanesi kullanılmıştır. ANTLR kütüphanesi ile sorgu kuralları tanımlanmış ve Java dilinde sorgu ayrıştırıcısı üretilmiştir. ANTLR temel olarak *Lexer* ve *Parser* bileşenlerinden oluşmaktadır. *Lexer* dil içerisinde kabul edilen karakterleri tanımlarken, *Parser* ise dilin mantıksal yapısını kurmak için kelime grupları arasındaki ilişkileri tanımlamaktadır. Örneğin basit bir toplama işlemini ele alalım:

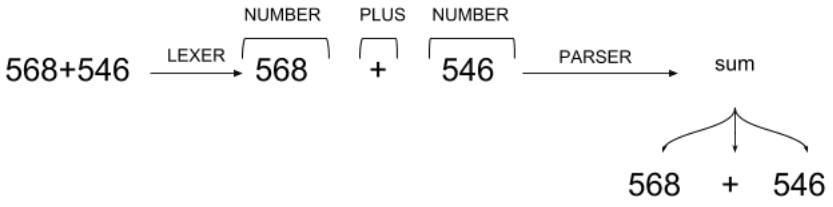
[11] ANother Tool for Language Recognition: http://www.antlr.org

```
SELECT q1.a1,q3.a1 FROM {
    <sub>Query1</sub>#documentdb as q1,
    <sub>Query2</sub>#keyvaluedb as q2,
    <sub>Query3</sub>#graphdb as q3,
    ...
} JOIN {
    q1.a1 = q2.a2
    AND
    q1.a1 = q3.a1
    ...
}
```

Şekil 5.3: PQL sorgu yapısı

$$568 + 546$$

Lexer, verilen metni tarayarak öncelikle '5', '6', '8' karakterlerini bulur ve ardından '+' karakteriyle karşılaşır. Dolayısıyla verilen ilk karakterlerin bir numarayı temsil ettiğini anlayabilir. Daha sonra gelen '+' karakterini bir operatör olarak tanımlayabilir ve ardından diğer numara grubunu bulabilir. *Parser* ise verilen numaralarla operatörün ilişkisini tanımlayabilir. *Lexer* ve *Parser'ın* bu işlevselliğe kavuşabilmesi için gramer dosyasında kuralların tanımlı olması gerekmektedir. Şekil 5.4'te yukarıdaki örneğin gramer dosyasında tanımlanan kurallara göre işleyiş mekanizması gösterilmiştir.




```
/*
 * Parser kuralı
 */
sum: NUMBER PLUS NUMBER;

/*
 * Lexer kuralları
 */
NUMBER: [0-9]+;
PLUS: '+';
```

Şekil 5.4: ANTLR toplama işlemi örneği

PQL'in temel iki *Parser* kuralı Algoritma 1 ve Algoritma 2'de verilmiştir.

Algoritma 1 *'select_stmt'* adlı *Parser* kuralıdır. Buna göre sorgu dili 'SELECT' anahtar kelimesiyle başlamalıdır. Ardından *'result_column'* ismine sahip virgülle ayrılabilen birden fazla kural alabilmektedir. *'result_column'* kuralı alt-sorguya verilen takma isim, nokta ve o veri modeli içerisinde sonuç olarak döndürülmesi istenilen değişken ismi olarak tanımlanabilmektedir. Alt-sorguların tanımlandığı kısım ise 'FROM' anahtar kelimesiyle diğer sözdiziminden ayrılmaktadır. 'FROM' anahtar kelimesinden sonra gelen süslü parantezlerin içerisinde birden fazla alt-sorgu *'subquery_stmt'* virgülle birbirlerinden ayrılarak tanımlanabilmektedir. *'select_stmt'* kuralında son olarak, birleşim işleminin tanımlanabilmektedir. Zorunlu olmayan bu alan 'JOIN' anahtar kelimesiyle başlayıp süslü parantezler içerisine birleşim kurallarının tanımlanmasıyla tarif edilebilmektedir. Birden fazla alt-birleştirme kuralı birbirinden 'AND' anahtar kelimesiyle ayrılabilmektedir. Alt-birleştirme kuralı şu şekilde tanımlanabilmektedir: alt-sorguya verilen takma isim, nokta, o veri modeli içerisinde birleştirilmek istenen değer, eşitlik değeri (=) ve yine eşitlik değerine kadar olan kısmın tekrarı. Bu durum basit bir biçimde şöyle ifade edilebilmektedir: $mongoDb.imdbId = redisDb.movieId$

Algoritma 1 Temel ayrıştırıcı kuralı

K_SELECT *result_column (',' result_column)**
K_FROM OPEN_CURLY_BRACKET *subquery_stmt (',' subquery_stmt)** CLOSE_CURLY_BRACKET
(K_JOIN OPEN_CURLY_BRACKET *subjoin_stmt (*K_AND *subjoin_stmt)** CLOSE_CURLY_BRACKET)?

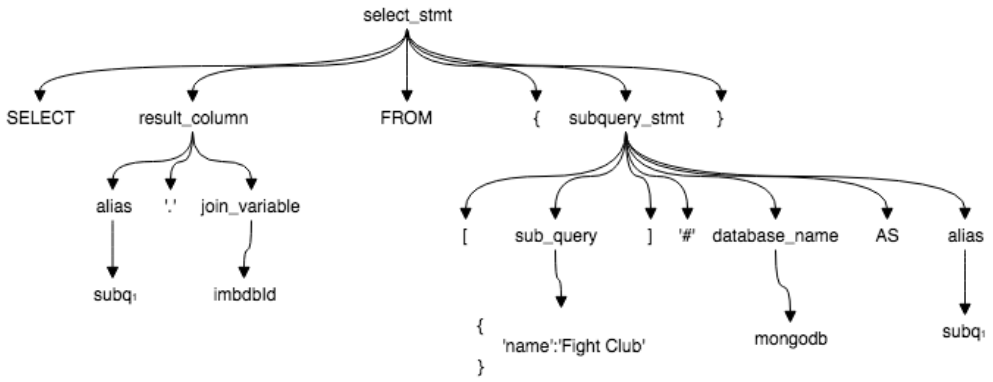
Algoritma 2 *'subquery_stmt'* adlı *Parser* kuralıdır. Bu kural alt-sorguların tanımlanabilmesini sağlamaktadır. Kurala göre bir alt-sorgu ifadesi köşeli parantezle başlamalı ve bitmelidir. etiketleri içerisine ise veri modeli üzerinde çalıştırılacak olan sorgu (*'sub_query'*) yazılmalıdır. Alt-sorgu ve sorgunun çalıştırılacağı veri modeli tanımı birbirinden *'#'* karakteriyle ayrılmaktadır. Veri modeli ismini ifade eden *'database_name'* kuralı bir kelime grubunu ifade etmektedir. Buraya kadar ifade edilen sözdizimi temsil etmek için 'AS' anahtar kelimesini takiben *'alias'* kuralıyla tanımlanmış olan kelime grupları kullanılabilmektedir. Örneğin: $< sub > \{'name' :' FightClub'\} < /sub >\#mongodbASsubq_1$

Algoritma 2 Alt-sorgu ayrıştırıcı kuralı

<sub>
  sub_query
</sub>
  '#' database_name K_AS *alias*

PQL'in örnek bir sorguyla ifade edilmiş soyut sözdizimi ağacı[12] gösterimi Şekil 5.5'teki gibidir.

[12] Abstract syntax tree (AST): https://en.wikipedia.org/wiki/Abstract_syntax_tree



Şekil 5.5: Soyut sözdizimi ağacı

5.2.1.2 Sorgu ayrıştıcı alt-birimin işlevselliği

Geliştirilen *Sorgu Ayrıştırıcı* alt-birimi kullanıcı tarafından PQL dili kullanılarak yazılmış olan sorguyu, ayrıştırmak ve *Sorgu Dağıtıcı* alt-birimine iletmekle sorumludur. PQL ile yazılmış sorgu birden fazla veri deposu üzerinde çalıştırılacak olabilir. _{İlgili veri deposu sorgusu}#veri-deposu-adı biçiminde tanımlanmış alt sorgular ayrıştırılır ve veri deposu bilgisi ile ilgili veri deposu sorgusu *Sorgu Dağıtıcı* alt-birimine iletilir. *Sorgu Ayrıştıcı*'nın çalışma biçimi Algoritma 3'te verilmiştir.

Algoritma 3 Sorgu ayrıştırıcı

Input: query Q

initialize pql *lexer* with *query*
initialize pql *parser* with *lexer*
context := parse Q
initialize pql *visitor*
assignment list := visit context // recursive visit
distribute(*assignment list*)

5.2.1.3 Sorgu dağıtıcı alt-birimin işlevselliği

Sorgu Ayrıştırıcı alt-birimi tarafından gönderilen veri deposu sorgusunu ve veri deposu bilgisini yakalayan *Sorgu Dağıtıcı* alt birimi, sorguları ilgili veri depoları üzerinde çalıştırmak için bu depolara sorguyu yönlendirmekle sorumludur. Bu alt-birim aynı zamanda sarmalayıcılar ile iletişim halindedir. Sorgu çalıştırma görevi sistemin veri deposuna sıkı-bağımlı yapısından dolayı sarmalayıcıların sorumlulukları altındadır. *Sorgu Dağıtıcı* gelen veri deposu bilgisi ile sorgunun çalıştırılacağı veri deposu sarmalayıcısını bulur ve sarmalayıcı üzerinde çalıştırma işlemini başlatmak için sarmalayıcılarla etkileşime girer. *Sorgu Dağıtıcı*'nın sorgu dağıtma işlemi Algoritma 4'te gösterilmiştir.

Algoritma 4 Sorgu dağıtıcı

Input: query assignment list QA
Output: integrated result
for *each assignment in QA* **do**
database name := get *database name* from *assignment*
wrapper:= findWith(*database name*)
result := executeTransform(get *query* from *assignment*)
integrate *result*
end

5.2.2 Sarmalayıcı katmanının geliştirimi

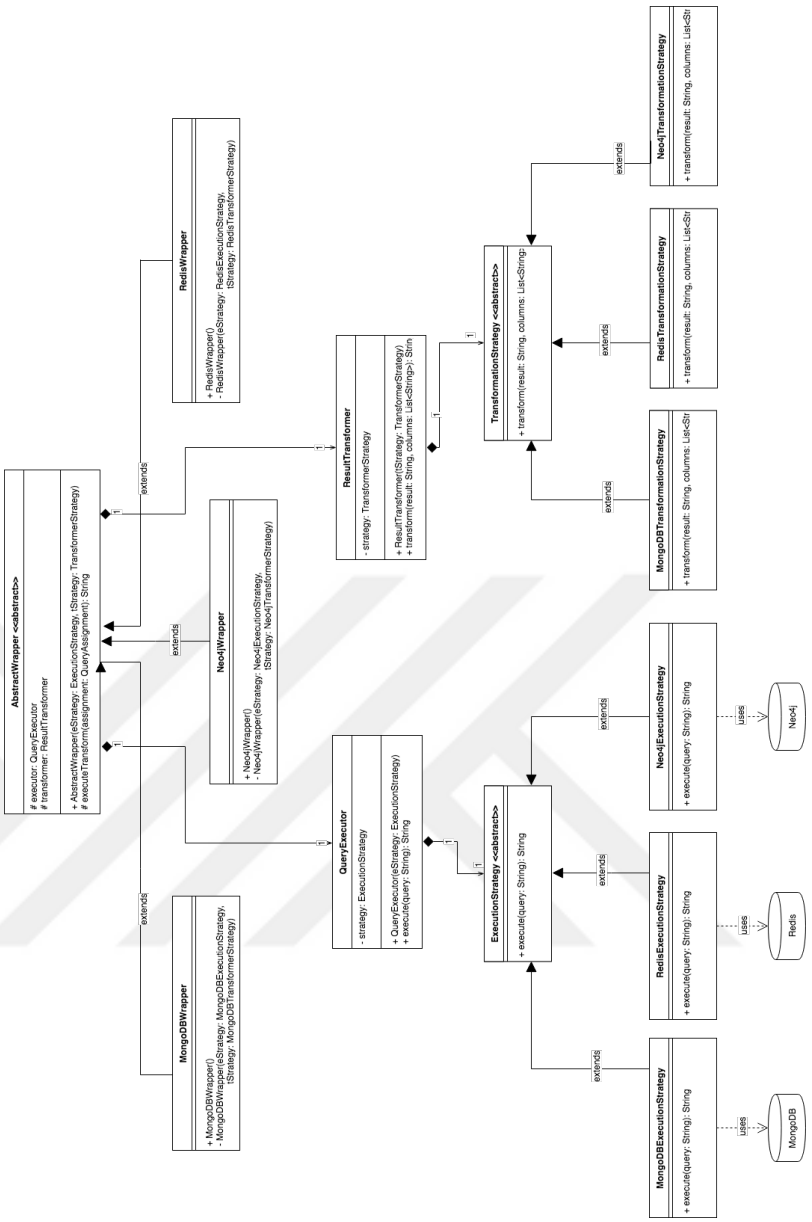
Sarmalayıcılar yerel depolar ile sistem arasındaki iletişimi ve bütünleştirmeyi sağlayan ara-birimlerdir. Sistemin sorgulama bakımından yerel depoya sıkı-bağımlı yapısı dolayısıyla her bir yerel depo için yazılmış bir sarmalayıcı bulunması gereklidir. Böylece sarmalayıcılara gelen sorgular yerel depolar üzerinde doğrudan çalıştırılmakta ve dönen sonuçların ortak modele (global şema) dönüşümü gerçekleştirilmektedir. Bu işlemi gerçekleştiren temel iki alt-birim mevcuttur: *Sorgu Çalıştırıcı* ve *Sonuç Dönüştürücü*.

Sarmalayıcılar Bölüm 6'te belirtilen kullanım durumuna göre Java yazılım dilinde geliştirilmiş olup UML şeması Şekil 5.6'te verilmiştir. Sarmalayıcı yapıların temel işlevselliği olan Sorgu Çalıştırma ve Sonuç Dönüştürme şema da görüleceği üzere strateji deseni ile geliştirilmiştir. Böylece soyut sınıf yapıları ve desenler kullanılarak basit, anlaşılır ve kolayca genişletilebilir bir tasarım ortaya koyulmuştur. Her bir sarmalayıcı yapı sorumlu olduğu depo ile ilişkili sorgu işletimi ve sonuç dönüşümü stratejilerini gerçekleştirmektedir. Sisteme entegre edilecek başka herhangi bir veri deposu için de sorgu çalıştırma ve sonuç dönüştürme stratejilerinin yazılması yeterlidir.

Örnek veri depoları olarak doküman için MongoDB, anahtar-değer için Redis ve çizge için Neo4j seçilmiştir. *Sorgu Dağıtıcı* alt-birim tarafından gelen sorgular *Sorgu Çalıştırıcı* alt-birim tarafından çalıştırılır ve sonuç *Sonuç Dönüştürücü* alt-birime iletilir. Sonuç sorgu dilinde *'result_column'* kuralında belirtilen değişkenlere göre dinamik olarak -kod bağımsız- dönüştürülür.

5.2.2.1 Sorgu çalıştırıcı alt-birimin işlevselliği

Sorgu Dağıtıcı tarafından sarmalayıcı arayüzüne gönderilen veri deposu sorgusu veri depolarının Java API'leri aracılığıyla *Sorgu Çalıştırıcı* alt-birimi



Şekil 5.6: Sarmalayıcı katmanına ait UML şeması

tarafından çalıştırılır. Şekil 5.6'teki UML şemasında da görülebileceği gibi *Sorgu Çalıştırıcı* alt-birimi *QueryExecutor* ismiyle tanımlanmış olup içerisinine bir *ExecutionStrategy* soyut sınıfını almaktadır. *ExecutionStrategy* herhangi bir veri deposunun sorgu çalıştırma stratejisi olabilmektedir. Örnek veri seti için seçilmiş olan doküman veri deposu MongoDB, *MongoDBExecutionStrategy* adıyla tanımlanmış olan sorgu çalıştırma stratejisine sahiptir. Bu strateji sorguyu alır ve veri deposu üzerinden çalıştırır. Daha sonra sonucu bir JSON dokümanı olarak döndürür. Bu sorgunun çalıştırılma sorumluluğu *QueryExecutor* sınıfına aittir. *Sorgu Çalıştırıcının* sorgu çalıştırma yapısı Algoritma 5'te açıklanmıştır.

Algoritma 5 Sorgu çalıştırıcı

Input: query Q
Output: results R

store := data stores client
result := query *store* with input Q
R := convert *results* to json format

5.2.2.2 Sonuç dönüştürücü alt-birimin işlevselliği

Algoritma 6 Sonuç dönüştürücü

Input: results as json R, result column variables V
Output: bindings list

```
set results to parse(R)
set bindings to empty
for each object in results do
   set bindMap to empty
   for each variable in V do
      put (variable, get variable in object) pair to bindMap
   end
   put bindMap to bindings list
end
```

Sorgu Çalıştırıcı çalıştırılan sorgunun sonucunu değer olarak döndürmektedir. *AbstractWrapper* sınıfı bu sorgu sonucunu almakta ve istenilen değerler listesiyle birlikte *Sonuç Dönüştürücü* alt-birime iletmekle sorumludur. Her bir veri deposu için yazılmış olan sonuç dönüştürücüler, sonuç içerisinde istelen değerleri yakalamakta ve bunları anahtar-değer veri yapısına sahip bellek içi bir yapıda saklamaktadırlar. Sonuç değerleri, *'result_column'* kuralında istenilen değerlere böylece eşlenebilmektedir. *Sonuç Dönüştürücü* alt-biriminin dinamik sonuç dönüştürme yapısı Algoritma 6'da açıklanmıştır.

5.2.2.3 Yeni veri deposu tanımlama adımları

Şekil 5.6'te görünen *QueryExecutor (Sorgu Çalıştırıcı)* soyut üst sınıfı, içerisinde sadece *execute* adında bir metod barındırmaktadır. Bu sınıftan türeyen diğer tüm veri deposu sorgu çalıştırıcıları bu metodun gerçekleştirimini kendi içerilerinde yapmak durumundadırlar. Yine şemada görünen *ResultTransformer (Sorgu Dönüştürücü)* soyut üst sınıfı içerisinde *transform* adında bir metod barındırmaktadır. Veri deposu sorgu dönüştürücülerinin görevi de bu metodun gerçekleştirimini kendi içlerinde yapmalarıdır. *AbstractWrapper (Sarmalayıcı)* soyut üst sınıfı kendi içerisinde *QueryExecutor* ve *ResultTransformer* stratejilerini barındırmaktadır. Sorgu dağıtıcı tarafından gelen veri deposu bilgisine göre bu

soyut üst sınıftan türeyen veri deposu sorgu çalıştırıcı ve sonuç dönüştürücüleri dinamik olarak yaratılmakta ve *AbstractWrapper*'ın *executeTransform* metodu sayesinde sorumluluklarını yerine getirmektedirler. Böylece sorgu dinamik olarak çalıştırılabilmektedir. Strateji deseniyle birlikte, sorgu çalıştırma ve sonuç dönüştürme aileleri tanımlanmakta ve her biri kapsüllenmektedir. Böylece aileler içersinde tanımlı sınıflar birbirleri yerine kullanılabilir hale gelmektedir. Strateji, algoritmanın onu kullanan istemcilerden bağımsız olarak değişmesine izin vermektedir. Veri deposu tanımla işleminin adımları şu şekilde tanımlanabilir:

1. Sorgu çalıştırıcı için *{Veri deposu ismi}ExecutionStrategy* isminde bir JAVA sınıfı yaratılmalı ve bu sınıf *ExecutionStrategy* soyut üst sınıfından türemelidir.

2. Tanımlanan yeni sınıf içerisinde soyut üst sınıfın *execute* metodunun gerçekleştirimi yapılmalıdır. Burada veri deposu üzerinde, parametre içerisinde gelen sorgu cümleciği doğrudan çalıştırıldıktan sonra sonuç JSON-String formatında veya String formatında döndürülmelidir.

3. Sonuç dönüştürücü için *{Veri deposu ismi}TransformationStrategy* isminde bir JAVA yaratılmalı ve bu sınıf *TransformerStrategy* soyut üst sınıfından türemelidir.

4. Tanımlanan yeni sınıf içerisinde soyut üst sınıfın *transform* metodunun gerçekleştirimi yapılmalıdır. Türetilen metod parametreleri içerisinde sorgu çalıştırma işlemi sonucu dönen sonuç ve PQL sorgusu içerisinde verilen ve sorgu sonucu dönülmesi istenilen değerler bulunmaktadır. Bu değerlere göre sorgu sonucu dolaşılmalı ve değerler aracılığıyla sonuçtan çekilen değerler eşlenmelidir. Bu işlem *BindingHashMap* sınıfı aracılığıyla gerçekleştirilmelidir.

5. Son olarak *AbstractWrapper* soyut sınıfından türetilmiş *{Veri deposu ismi}Wrapper* oluşturulmalı ve sınıf içerisinde birisi özel (private) bir diğeri de genel (public) olmak üzere iki ilklendirici yaratılmalıdır. Özel ilklendirici ilk parametre olarak *{Veri deposu ismi}ExecutionStrategy*'i almalıdır. İkinci parametre olarak da *{Veri deposu ismi}TransformationStrategy* verilmelidir. Genel ilklendirici parametre almamalı ve içerisinde özel ilklendiriciyi içermelidir.

Yaratılan yeni sarmalayıcı çoklu depo sistemine *WrapperFactory* sınıfı içerisinde genel ilklendirici ile yaratılarak tanımlanabilmektedir.

5.3 Prototip Çoklu Depo Sisteminin Literatürdeki Benzer Sistemlerle Karşılaştırılması

Çoklu depo sistemleri Bondiombouy ve Valduriez'in (2016) belirttiği gibi Bölüm 4.8'de sınıflandırılmıştır. Geliştirilen prototip çoklu depo sistemi de bu sınıflandırmaya dahil edilebilmektedir. Tablo 5.1'te prototip çoklu depo sisteminin işlevselliği, Tablo 5.2'te ise gerçekleştirimi incelenmiştir.

Çoklu depo sistemi	Hedef	Veri Modeli	Sorgu Dili	Veri Depoları
Prototip çoklu depo sistemi	NoSQL verilerini sorgulama	Anahtar-değer	PQL	NoSQL

Çizelge 5.1: Prototip çoklu depo sisteminin işlevselliği

Çoklu depo sistemi	Özel Birimler	Şemalar	Sorgu İşleme	Sorgu Eniyileştirme
Prototip çoklu depo sistemi	Sorgu ayrıştırıcı, sorgu dağıtıcı, sorgu çalıştırıcı, sonuç dönüştürücü	GAV	Veri deposu yetenekleri	Yok

Çizelge 5.2: Prototip çoklu depo sisteminin gerçekleştirim teknikleri

Diğer yandan çoklu depo sisteminin esnek mimari yapısı da diğer araçlarla olan karşılaştırılmaya dahil edilebilmektedir. Örneğin geliştirilen ESTOCADA aracı çok kolay genişleyebilir bir mimariye sahip değildir. Dolayısıyla ortaya koydukları senaryonun sorgulama ihtiyaçlarının değişmesiyle birlikte mimari yapı buna ayak uyduramamış; yeniden yazma ve veriyi yeniden taşıma maliyetleri ortaya çıkmıştır (Bugiotti et al., 2016). Bunu gerçekleştirebilecek insan gücü yetersizliğinden dolayı da süreç sonlandırılmıştır. Ortaya koyulan prototip çoklu depo sistemi kolayca genişleyebilen, modüler sarmalayıcı alt yapısına sahiptir: Sisteme yeni bir sarmalayıcı ve daha doğru bir ifadeyle veri deposu entegre işlemi oldukça basittir. Diğer yandan sorguların çalıştırılması veri depolarının kendi dilleriyle gerçekleştirildikleri için karmaşık sorgulara refleks üretememe gibi bir problemle de karşı karşıya kalınmamaktadır. Sorgu çalıştırma görevi veri deposunun sorgu motoruna ait bir sorumluluk olarak bırakılmıştır. Yazılan sarmalayıcıların görevi ise veri deposu sorgu motoruna sorguyu iletmek ve dönen sonucu uygun formata dönüştürmek olarak belirlenmiştir.

Bölüm 4'te anlatıldığı gibi kimi literatür çalışmaları, SQL benzeri bir dil oluşturmakta bu dilden veri deposu dillerine dönüşüm sağlayarak sorgu çalıştırma işlemini gerçekleştirmektedir. Ancak bu tür dönüşümler karmaşık sorgularda istenildiği gibi çalışmamakta ve doğru sorgu veri deposuna iletilememektedir. Diğer yandan sorgu dönüşümleri, sorgu çalıştırma zamanı bakımından kullanıcıya kayıp

yaşatmaktadır. Geliştirilen PQL sorgulama dili ile birlikte SQL benzeri ancak ayrı ayrı veri depoları sorgularını içerisinde barındıran bir yapı ortaya koyulmuştur. Böylece sorgu dönüşüm maliyetlerinden kurtulunmuş ve karmaşık sorguları da işletebilir bir tasarım elde edilmiştir.

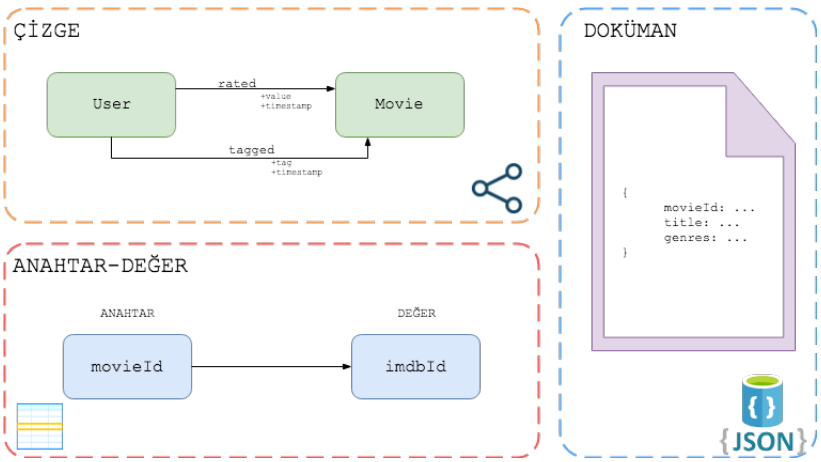
6 ÇOKLU DEPO SİSTEMİNİN DEĞERLENDİRİLMESİ

Sarmalayıcı katmanının geliştirimi için örnek bir veri kümesi belirlenmiş ve bu veri kümesi üzerinden bir çoklu depo kullanım durumu yaratılmıştır. Başlangıçta MovieLens'in[13] eğitim ve geliştirim için sunmuş olduğu küçük veri kümesi seçilmiştir. Bu veri kümesinin içeriği filmler (movies), bağlatılar (links), puanlar (ratings) ve etiketler (tags) biçiminde olup dosyalarda sunulan içerikler Tablo 6.1'de gösterilmiştir.

movies.csv	**links.csv**	**ratings.csv**	**tags.csv**
movieId	movieId	userId	userId
title	imdbId	movieId	movieId
genres	tmdbId	rating	tag
		timestamp	timestamp

Çizelge 6.1: MovieLens veri kümesi içeriği

Çoklu depo sistemi için yaratılan üç modelli kullanım durumu Şekil 6.1'de görülmektedir. Çizge veritabanında kullanıcıların filmleri oylama ve etiketleme bilgileri yer almaktadır. Çizge veritabanının içeriği Tablo 6.1'de gözüken tags.csv ve ratings.csv dosyalarından alınmıştır. Anahtar-değer veritabanında links.csv dosyasında bulunan movieId'lerinin imdbId karşılıkları yer almaktadır. Son olarak doküman veritabanında da filmler hakkındaki genel bilgiler (id, başlık, tür vb.) saklanmaktadır. Bu bilgiler de movies.csv dosyasından alınmıştır.



Şekil 6.1: MovieLens veri kümesi çoklu depo kullanım durumu

Depolara veriler yazılan bir Java uygulamasıyla aktarıldıktan sonra kullanım durumu oluşturabilecek sorgular ortaya çıkarılmıştır. Kullanım durumu için doküman veritabanı olarak MongoDB, anahtar-değer veritabanı olarak

[13]MovieLens veri kümeleri: https://grouplens.org/datasets/movielens/

Redis ve son olarak çizge veritabanı olarak da Neo4j seçilmiştir. Bu üç farklı veritabanı için sarmalayıcılar yazılmış; MongoDB için sorgu çalıştırıcı strateji olan *MongoDBExecutionStrategy* ve sonuç dönüştürücü strateji olan *MongoDBTransformerStrategy* yazılmıştır. Aynı şekilde diğer veritabanları için *RedisExecutionStrategy, RedisTransformerStrategy, Neo4jExecutionStrategy* ve *Neo4jTransformerStrategy* gerçekleştirimleri yapılmıştır.

6.1 Birinci Seviye Sorgu Örnekleri

Kullanım durumu için öncelikle tek tek veri modelleri sorgulanmıştır: MongoDB için başlığı verilen filmin movieId'si istenmiştir.

Örnek sorgu:

```
SELECT doc.movieId FROM{
  <sub>{'title':"Fight Club (1999)"}</sub>#mongodb AS doc
}
```

Sorgu sonucu:

```
{
  "head": {
    "vars": [ "movieId" ]
  } ,
  "results": {
    "bindings": [
      {
        "movieId": { "type": "literal" , "value": "2959" }
      }
    ]
  }
}
```

Redis için movieId'si verilen bir filmin imdbId karşılığı istenmiştir.

Örnek sorgu:

```
SELECT keyval.imdbId FROM{
  <sub>2959</sub>#redis ASkeyval
}
```

Sorgu sonucu:

```
{
   "head": {
      "vars": [ "imdbId" ]
   } ,
   "results": {
      "bindings": [
         {
            "imdbId": { "type": "literal" , "value": "0137523" }
         }
      ]
   }
}
```


Neo4j için movieId'si verilen bir filme kullanıcılar tarafından yapılan etiket bilgileri istenmiştir.

Örnek sorgu:

```
SELECT graph.userId, graph.value, graph.timestamp FROM{
   [MATCH (n1:User)-[r:TAGGED]->(n2:Movie) WHERE n2.movieId=\"2959\" RETURN n1.userId as userId, r.value as
value, r.timestamp as timestamp"]#neo4j AS graph
}
```

Sorgu sonucu:


```
{
   "head": {
      "vars": [ "userId" , "value" , "timestamp" ]
   } ,
   "results": {
      "bindings": [
         {
            "userId": { "type": "literal" , "value": "501" } ,
            "value": { "type": "literal" , "value": "twist ending" } ,
            "timestamp": { "type": "literal" , "value": "1292956479" }
         } ,
         {
            "userId": { "type": "literal" , "value": "501 } ,
            "value": {"type": "literal" , "value": "psychology" } ,
            "timestamp": { "type": "literal" , "value": "1292956476" }
         } ,
         {
            "userId": { "type": "literal" , "value": "501" } ,
            "value": { "type": "literal" , "value": "dark comedy" } ,
            "timestamp": { "type": "literal" , "value": "1292956481" }
         }
      ]
   }
}
```

6.2 İkinci Seviye Sorgu Örnekleri

İkinci seviye sorguların ilk örneğini MongoDB ve Redis veritabanlarına atılan ortak sorgu oluşturmaktadır. Sorguda movieId'si verilen bir film verisine ait başlık ve imdbId alanları istenmiştir.

Örnek sorgu:

```
SELECT doc.title, keyval.imdbId FROM {
  <sub>{'movieId':3}</sub>#mongodb AS doc,
  <sub>3</sub>#redis AS keyval
};
```

Sorgu sonucu:


```
[
  {
    "head":{
      "vars":["title"]
    },
    "results":{
      "bindings":[
        {
          "title":{"type":"literal","value":"Grumpier Old Men (1995)"}
        }
      ]
    }
  },
  {
    "head":{
      "vars":["imdbId"]
    },
    "results":{
      "bindings":[
        {
          "imdbId":{"type":"literal","value":"0113228"}
        }
      ]
    }
  }
]
```


Bir sonraki örneği MongoDB ve Neo4j veritabanlarına atılan ortak sorgu oluşturmaktadır. Bu sorguda movieId'si verilen bir film verisine ait başlık, tür ve kullanıcılar tarafından bu filme ait yapılan oylamalar listelenmiştir.

Örnek sorgu:

```
SELECT doc.title, doc.genres, graph.userId, graph.value FROM {
   <sub>{'movieId':4}</sub>#mongodb AS doc,
   <sub>MATCH (n1:User)-[r:RATED]->(n2:Movie) WHERE n2.movieId="4" RETURN n1.userId as userId, r.value as
value</sub>#neo4j AS graph
};
```

Sorgu sonucu:


```
[
   {
      "head":{
         "vars":["title","genres"]
      },
      "results":{
         "bindings":[
            {
               "title":{"type":"literal","value":"Waiting to Exhale (1995)"},
               "genres":{"type":"literal","value":"Comedy|Drama|Romance"}
            }
         ]
      }
   },
   {
      "head":{
         "vars":["userId","value"]
      },
      "results":{
         "bindings":[
            {"userId":{"type":"literal","value":"518"},"value":{"type":"literal","value":"1.0"}},
            {"userId":{"type":"literal","value":"113"},"value":{"type":"literal","value":"3.0"}},
            {"userId":{"type":"literal","value":"128"},"value":{"type":"literal","value":"3.0"}},
            {"userId":{"type":"literal","value":"19"},"value":{"type":"literal","value":"3.0"}},
            {"userId":{"type":"literal","value":"461"},"value":{"type":"literal","value":"1.5"}},
            {"userId":{"type":"literal","value":"460"},"value":{"type":"literal","value":"3.5"}},
            {"userId":{"type":"literal","value":"391"},"value":{"type":"literal","value":"2.0"}},
            {"userId":{"type":"literal","value":"239"},"value":{"type":"literal","value":"3.0"}},
            {"userId":{"type":"literal","value":"182"},"value":{"type":"literal","value":"3.0"}},
            {"userId":{"type":"literal","value":"168"},"value":{"type":"literal","value":"3.0"}},
            {"userId":{"type":"literal","value":"644"},"value":{"type":"literal","value":"1.0"}},
            {"userId":{"type":"literal","value":"649"},"value":{"type":"literal","value":"3.0"}},
            {"userId":{"type":"literal","value":"650"},"value":{"type":"literal","value":"1.0"}}
         ]
      }
   }
]
```


Son olarak, Redis ve Neo4j veritabanlarına atılan ortak sorgu işletilmiştir. Sorguda movieId'si verilen bir filme ait imdbId istenmiş ve hem filmi etiketlemiş hem de filme oy vermiş kullanıcı id'leri listelenmiştir.

Örnek sorgu:

```
SELECT keyval.imdbId, graph.userId FROM {
  <sub>260</sub>#redis ASkeyval,
  <sub>MATCH (n1:User)-[r1:RATED]->(n2:Movie)<-[r2:TAGGED]-(n1:User) WHERE n2.movieId="260" RETURN
DISTINCT n1.userId as userId</sub>#neo4j AS graph
};
```

Sorgu sonucu:


```
[
  {
    "head":{
      "vars":["imdbId"]
    },
    "results":{
      "bindings":[
        {"imdbId":{"type":"literal","value":"0076759"}}
      ]
    }
  },
  {
    "head":{
      "vars":["userId"]
    },
    "results":{
      "bindings":[
        {"userId":{"type":"literal","value":"200"}},
        {"userId":{"type":"literal","value":"431"}},
        {"userId":{"type":"literal","value":"663"}},
        {"userId":{"type":"literal","value":"179"}},
        {"userId":{"type":"literal","value":"630"}},
        {"userId":{"type":"literal","value":"138"}},
        {"userId":{"type":"literal","value":"314"}},
        {"userId":{"type":"literal","value":"402"}},
        {"userId":{"type":"literal","value":"448"}},
        {"userId":{"type":"literal","value":"503"}},
        {"userId":{"type":"literal","value":"481"}},
        {"userId":{"type":"literal","value":"660"}}
      ]
    }
  }
]
```


6.3 Üçüncü Seviye Sorgu Örnekleri

Son olarak sistemin çalışabilirliğinin test edilebilmesi için çoklu sorgu örneği çalıştırılmıştır. Çoklu sorgu aşağıdaki gibidir:

Örnek sorgu:

```
SELECT doc.movieId, keyval.imdbId, graph.userId, graph.value, graph.timestamp FROM {
  <sub>{'title':"Fight Club (1999)"}</sub>#mongodb AS doc,
  <sub>2959</sub>#redis AS keyval,
  <sub>MATCH (n1:User)-[r:TAGGED]->(n2:Movie) WHERE n2.movieId='2959' RETURN n1.userId as userId, r.value
as value, r.timestamp as timestamp</sub>#neo4j AS graph
};
```

Sorgu sonucu:


```
[
  {
    "head": {
      "vars": [ "movieId" ]
    },
    "results": {
      "bindings": [
        {
          "movieId": { "type": "literal", "value": "2959" }
        }
      ]
    }
  },
  {
    "head": {
      "vars": [ "imdbId" ]
    },
    "results": {
      "bindings": [
        {
          "imdbId": { "type": "literal", "value": "0137523" }
        }
      ]
    }
  },
  {
    "head": {
      "vars": [ "userId", "value", "timestamp" ]
    },
    "results": {
      "bindings": [
        {
          "userId": { "type": "literal", "value": "501" },
          "value": { "type": "literal", "value": "twist ending" },
          "timestamp": { "type": "literal", "value": "1292956479" }
        },
        {
          "userId": { "type": "literal", "value": "501" },
          "value": { "type": "literal", "value": "psychology" },
          "timestamp": { "type": "literal", "value": "1292956476" }
        },
```

```
          {
            "userId": { "type": "literal", "value": "501" },
            "value": { "type": "literal", "value": "dark comedy" },
            "timestamp": { "type": "literal", "value": "1292956481" }
          }
        ]
      }
    }
]
```

7 SONUÇ VE TARTIŞMA

Artan Web kullanımı, akıllı cihazların üretim sürecine dahil olması ve bulut tabanlı uygulamaların gelişimi birbirinden farklı formatta ve büyük miktarlarda verinin tek bir uzayda fakat farklı konumlarda birikmesine yol açmıştır. Verilerin biriktirildiği bu konumlar yapısal olarak da birbirinden ayrılmaktadır. Çünkü "herkese uygun tek bir çözüm yok" mottosundan hareketle verilerin tümü tek bir veri deposunda depolanmamakta, ihtiyaca ve veri tipine yönelik geliştirilen NoSQL veri depoları bu soruna çözüm olarak ortaya çıkmaktadır. Ancak bu durum da veri erişim ve entegrasyon problemlerine neden olmaktadır. Erişim problemi; veri depolarına erişilecek tek bir arayüz olmamasından kaynaklanma birlikte entegrasyon problemi; verilerin farklı formatlarda depolanması ve farklı konumlardan bulunmasından ve bununla birlikte veri depoları sorgulayabilecek ortak bir arayüz olmamasından kaynaklanmaktadır. Bu tezle birlikte geliştirilen prototip çoklu depo sistemi aracılığıyla bu sorunlara çözüm bulunması hedeflenmiştir.

Prototip çoklu depo sistemi veriye tek bir yerden erişim problemine geliştirilmiş olan arabulucu-sarmalayıcı mimari aracılığıyla çözüm getirmektedir. Yazılan sarmalayıcılar depoların var olan JAVA API'lerini kullanarak veri depolarına uygulama içerisinden kolayca erişebilmektedirler. Ayrıca geliştirilen soyut sarmalayıcı arayüzüyle birlikte sisteme yeni veri deposu kolayca entegre edilebilmektedir. Böylece sistemin kullanıcıları verilerinin saklandığı veri depolarını sisteme entegre ederek sorgulama işlemlerini gerçekleştirebilmektedirler. Diğer yandan, veri entegrasonu problemine çözüm olarak da sistemin ortak bir arayüzden sorgulanabilmesi için arabulucu katmanında PQL adında bir sorgulama dili geliştirilmiş ve sorgunun ayrıştırılıp farklı veri depoları üzerinde çalıştırılması için buna uygun ayrıştırıcı geliştirimi yapılmıştır. Böylece bir sorgu dili aracılığıyla sisteme entegre veri depoları sorgulanabilmekte ve sorgu sonuçları alınabilmektedir.

Çoklu depo sisteminin prototip düzeyinde kalmasının iki önemli bileşeni vardır: Bunlardan ilki, sonuç birleştirme işleminin karmaşık algoritmalarla temsil edilmemesi ve ikincisi, sistemin tek bir makine üzerinde çalışır durumda olmasıdır. Sonuç birleştirme basit bir biçimde, veri depolarına atılan sonuçların, bir veri yapısı aracılığıyla arda arda eklenmesiyle olmaktadır. Ancak burada veri depolarının entegrasyonun hangi değişkenler üzerinden yapılacağının sorgu içerisinde temsil edilmesi - ki PQL buna olanak vermektedir - ve dönen sonuçların bu değişkenler üzerinden birleştirilerek oluşturulması gerekmektedir. Diğer yandan ölçeklenebilirliği sağlamak için çoklu depo sistemi dağıtık çalışabilirliği

desteklemelidir. Ancak şu an için bunun geliştirimi yapılmamıştır ve geliştirilmiş olan sistem tek bir makine üzerinde hizmet verir durumdadır.

Çoklu depo sistemleri bilgisayar bilimleri alanında yeni yeni tartışılan ve açık kaynak kodlu herhangi bir uygulama örneği olmayan bir alandır. 4 bölümünde özetlendiği üzere araçların sistem mimarileri anlatılmıştır ancak sistemlerin çalışmasına yönelik performans metrikleri vb. henüz yayınlanmamıştır. Bundan sonraki çalışmayı; sonuç birleştirme algoritmalarının entegrasyonu, sistemin dağıtık bir mimari üzerinde sunulması ve daha sonra performans metriklerinin ölçülmesi oluşturacaktır.

KAYNAKLAR DİZİNİ

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A.: 2009, Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads, *Proceedings of the VLDB Endowment* **2(1)**, 922

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al.: 2015, Spark sql: Relational data processing in spark, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, 1383–1394 pp

Bondiombouy, C., Kolev, B., Levchenko, O., and Valduriez, P.: 2015, Integrating big data and relational data with a functional sql-like query language, in *International Conference on Database and Expert Systems Applications*, Springer, 170–185 pp

Bondiombouy, C., Kolev, B., Levchenko, O., and Valduriez, P.: 2016, Multistore big data integration with cloudmdsql, in *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII*, Springer, 48–74 pp

Bondiombouy, C. and Valduriez, P.: 2016, Query processing in multistore systems: an overview, *International Journal of Cloud Computing* **5(4)**, 309

Brewer, E.: 2012, Cap twelve years later: How the" rules" have changed, *Computer* **45(2)**, 23

Brewer, E. A.: 2000, Towards robust distributed systems, in *PODC*, Vol. 7

Bugiotti, F., Bursztyn, D., Deutsch, A., Ileana, I., and Manolescu, I.: 2015a, Invisible glue: scalable self-tuning multi-stores, in *Conference on Innovative Data Systems Research (CIDR)*

Bugiotti, F., Bursztyn, D., Deutsch, A., Ileana, I., and Manolescu, I.: 2015b, Toward scalable hybrid stores, in *SEBD Italian Symposium on Advanced Database Systems*

Bugiotti, F., Bursztyn, D., Deutsch, A., Manolescu, I., and Zampetakis, S.: 2016, Flexible hybrid stores: Constraint-based rewriting to the rescue, in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, IEEE, 1394–1397 pp

Chen, P., Gadepally, V., and Stonebraker, M.: 2016, The bigdawg monitoring framework, in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, IEEE, 1–6 pp

Dasgupta, S., Coakley, K., and Gupta, A.: 2016, Analytics-driven data ingestion

KAYNAKLAR DİZİNİ (devam)

and derivation in the awesome polystore, in *Big Data (Big Data), 2016 IEEE International Conference on*, IEEE, 2555–2564 pp

DeWitt, D. J., Halverson, A., Nehme, R., Shankar, S., Aguilar-Saborit, J., Avanes, A., Flasza, M., and Gramling, J.: 2013, Split query processing in polybase, in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ACM, 1255–1266 pp

Duggan, J., Elmore, A. J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., and Zdonik, S.: 2015, The bigdawg polystore system, *ACM Sigmod Record* **44(2)**, 11

Dziedzic, A., Duggan, J., Elmore, A. J., Gadepally, V., and Stonebraker, M.: 2015, Bigdawg: a polystore for diverse interactive applications, in *Data Syst. Interactive Anal. Workshop 2015*

Dziedzic, A., Elmore, A. J., and Stonebraker, M.: 2016, Data transformation and migration in polystores, in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, IEEE, 1–6 pp

Elmasri, R. and Navathe, S.: 2010, *Fundamentals of database systems*, Addison-Wesley Publishing Company

Elmore, A., Duggan, J., Stonebraker, M., Balazinska, M., Cetintemel, U., Gadepally, V., Heer, J., Howe, B., Kepner, J., Kraska, T., et al.: 2015, A demonstration of the bigdawg polystore system, *Proceedings of the VLDB Endowment* **8(12)**, 1908

Gadepally, V., Chen, P., Duggan, J., Elmore, A., Haynes, B., Kepner, J., Madden, S., Mattson, T., and Stonebraker, M.: 2016, The bigdawg polystore system and architecture, in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, IEEE, 1–6 pp

Gadepally, V., O'Brien, K., Dziedzic, A., Elmore, A., Kepner, J., Madden, S., Mattson, T., Rogers, J., She, Z., and Stonebraker, M.: 2017, Bigdawg version 0.1, in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, IEEE, 1–7 pp

Gilbert, S. and Lynch, N.: 2002, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, *Acm Sigact News* **33(2)**, 51

Gupta, A. M., Gadepally, V., and Stonebraker, M.: 2016, Cross-engine query execution in federated database systems, in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, IEEE, 1–6 pp

KAYNAKLAR DİZİNİ (devam)

Han, J., Haihong, E., Le, G., and Du, J.: 2011, Survey on nosql database, in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, IEEE, 363–366 pp

He, C.: 2015, Survey on nosql database technology, *Journal of Applied Science and Engineering Innovation Vol* 2(2)

Inc., M.: 2015, *Top 5 Considerations When Evaluating NoSQL Databases*, https://www.mongodb.com/white-papers, (Erişim tarihi: 13 Mayıs 2018)

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., and Gosain, D.: 2012, A survey and comparison of relational and non-relational database, *International Journal of Engineering Research & Technology* 1(6)

Kolev, B., Bondiombouy, C., Levchenko, O., Valduriez, P., Jimenez-Péris, R., Pau, R., and Pereira, J.: 2016a, Design and implementation of the cloudmdsql multistore system, in *CLOSER: Cloud Computing and Services Science*, Vol. 1, 352–359 pp

Kolev, B., Bondiombouy, C., Valduriez, P., Jiménez-Peris, R., Pau, R., and Pereira, J.: 2016b, The cloudmdsql multistore system, in *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2113–2116 pp

Kolev, B., Bondiombouy, C., Valduriez, P., Jiménez-Peris, R., Spain, R., and Pereira, J.: 2016c, Demonstration of the cloudmdsql multistore system, in *BDA: Bases de Données Avancées*

Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R., and Pereira, J.: 2016d, Cloudmdsql: querying heterogeneous cloud data stores with a common language, *Distributed and parallel databases* **34(4)**, 463

Leavitt, N.: 2010, Will nosql databases live up to their promise?, *Computer* 43(2)

Lenzerini, M.: 2002, Data integration: A theoretical perspective, in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ACM, 233–246 pp

Maccioni, A., Basili, E., and Torlone, R.: 2016, Quepa: Querying and exploring a polystore by augmentation, in *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2133–2136 pp

Moniruzzaman, A. and Hossain, S. A.: 2013, Nosql database: New era of databases for big data analytics-classification, characteristics and comparison, *arXiv preprint arXiv:1307.0191*

Nayak, A., Poriya, A., and Poojary, D.: 2013, Type of nosql databases and

KAYNAKLAR DİZİNİ (devam)

its comparison with relational databases, *International Journal of Applied Information Systems* **5(4)**, 16

Ong, K. W., Papakonstantinou, Y., and Vernoux, R.: 2014a, The sql++ query language: Configurable, unifying and semi-structured, *arXiv preprint arXiv:1405.3631*

Ong, K. W., Papakonstantinou, Y., and Vernoux, R.: 2014b, The sql++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases, *CoRR, abs/1405.3631*

Ong, K. W., Papakonstantinou, Y., and Vernoux, R.: 2014c, The sql++ unifying semi-structured query language, and an expressiveness benchmark of sql-on-hadoop, nosql and newsql databases, *CoRR, abs/1405.3631*

Özsu, M. T. and Valduriez, P.: 2011, *Principles of distributed database systems*, Springer Science & Business Media

Padhy, R., Ranjan, M., Suresh, P., Satapathy, C., and India, O.: 2018, Rdbms to nosql: Reviewing some next-generation non-relational database's

Papakonstantinou, Y.: 2016, Polystore query rewriting: The challenges of variety., in *EDBT/ICDT Workshops*

Podkorytov, M., Soderman, D., and Gubanov, M.: 2017, Hybrid. poly: An interactive large-scale in-memory analytical polystore, in *Data Mining Workshops (ICDMW), 2017 IEEE International Conference on*, IEEE, 43–50 pp

Pokorny, J.: 2013, Nosql databases: a step to database scalability in web environment, *International Journal of Web Information Systems* **9(1)**, 69

Pritchett, D.: 2008, Base: An acid alternative, *Queue* **6(3)**, 48

Sadalage, P. J. and Fowler, M.: 2012, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*, Pearson Education

Saeed, M., Villarroel, M., Reisner, A. T., Clifford, G., Lehman, L.-W., Moody, G., Heldt, T., Kyaw, T. H., Moody, B., and Mark, R. G.: 2011, Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database, *Critical care medicine* **39(5)**, 952

She, Z., Ravishankar, S., and Duggan, J.: 2016, Bigdawg polystore query optimization through semantic equivalences, in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, IEEE, 1–6 pp

Simitsis, A., Wilkinson, K., Castellanos, M., and Dayal, U.: 2012, Optimizing

KAYNAKLAR DİZİNİ (devam)

analytic data flows for multiple execution engines, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ACM, 829–840 pp

Stonebraker, M.: 2010, Sql databases v. nosql databases, *Communications of the ACM* **53(4)**, 10

Stonebraker, M. and Cetintemel, U.: 2005, "one size fits all": an idea whose time has come and gone, in *21st International Conference on Data Engineering (ICDE'05)*, 2-11 pp

Strauch, C., Sites, U.-L. S., and Kriha, W.: 2011, Nosql databases, *Lecture Notes, Stuttgart Media University* 20

Vogels, W.: 2009, Eventually consistent, *Communications of the ACM* **52(1)**, 40

Wang, G. and Tang, J.: 2012, The nosql principles and basic application of cassandra model, in *Computer Science & Service System (CSSS), 2012 International Conference on*, IEEE, 1332–1335 pp

Wiederhold, G.: 1992, Mediators in the architecture of future information systems, *Computer* **25(3)**, 38

Zhu, M. and Risch, T.: 2011, Querying combined cloud-based and relational databases, in *Cloud and Service Computing (CSC), 2011 International Conference on*, IEEE, 330–335 pp

ÖZGEÇMİŞ

Adı Soyadı: Bahtiyar ERDEN

Doğum Tarihi: 1990

Doğum Yeri: Uşak

Uyruğu: T.C.

EĞİTİM

Lisans: Ege Üniversitesi
Bilgisayar Mühendisliği Bölümü, 2013

Lise: Karataş Lisesi, İzmir, 2008

YAYINLAR

Halaç, T. G., Erden, B., İnan, E., Oğuz, D., Göçebe, P. and Dikenelli, O.: 2013, Publishing and linking university data considering the dynamism of datasources., in *Proceedings of the 9th International Conference on Semantic Systems*, ACM, 140-145 pp

Halaç, T. G., Erden, B., İnan, E., Oğuz, D., Göçebe, P. and Dikenelli, O.: 2013, Bağlı Veri Teknolojileri Kullanılarak Üniversite Verisinin Bütünleştirilmesi ve Yayınlanması, in *UYMS'13 7.Ulusal Yazılım Mühendisliği Sempozyumu*