# SEMANTİK VERİLERİN DAĞITIK ORTAMDA ETKİN OLARAK DEPOLANMASI VE SORGULANMASI

ÖZGÜR EROĞLU

YÜKSEK LİSANS TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

ARALIK 2013

ANKARA

Fen Bilimleri Enstitü onayı

_____

Prof. Dr. Necip CAMUŞCU

Müdür

Bu tezin Yüksek Lisans derecesinin tüm gereksinimlerini sağladığını onaylarım.

_____

Doç. Dr. Erdoğan DOĞDU

Anabilim Dalı Başkanı

ÖZGÜR EROĞLU tarafından hazırlanan SEMANTİK VERİLERİN DAĞITIK ORTAMDA ETKİN OLARAK DEPOLANMASI VE SORGULANMASI adlı bu tezin Yüksek Lisans tezi olarak uygun olduğunu onaylarım.

_____

Doç. Dr. Erdoğan DOĞDU

Tez Danışmanı

Tez Jüri Üyeleri

Başkan : Doç. Dr. Osman ABUL _____

Üye : Doç. Dr. Erdoğan DOĞDU _____

Üye : Doç. Dr. Kadir ERTOĞRAL _____

## TEZ BİLDİRİMİ

Tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, ayrıca tez yazım kurallarına uygun olarak hazırlanan bu çalışmada orijinal olmayan her türlü kaynağa eksiksiz atıf yapıldığını bildiririm.

Özgür EROĞLU

**Üniversitesi** : **TOBB Ekonomi ve Teknoloji Üniversitesi**
**Enstitüsü** : **Fen Bilimleri**
**Anabilim Dalı** : **Bilgisayar Mühendisliği**
**Tez Danışmanı** : **Doç. Dr. Erdoğan DOĞDU**
**Tez Türü ve Tarihi** : **Yüksek Lisans – Aralık 2013**

**Özgür EROĞLU**

# SEMANTİK VERİLERİN DAĞITIK ORTAMDA ETKİN OLARAK DEPOLANMASI VE SORGULANMASI

## ÖZET

İnternetin sağlamış olduğu en önemli özelliklerden bir tanesi, bilgilerin birbirine bağlanabilir olmasıdır. Bu sayede, birbiri ile ilişkili içerikler, veriyi yayınlayan kişilerden veya kaynaklardan bağımsız olarak, bir ağ biçiminde ulaşılabilir olmaktadır. Ancak, internet ortamındaki verinin çokluğu ve çeşitliliği, kullanıcıların ihtiyaç duydukları bilgilere hızlı ulaşmalarını ve bu veriyi hızlı işlemelerini zorlaştırmaktadır. Verinin bilgisayarlar tarafından anlaşılabilir ve yorumlanabilir olması durumunda, kullanıcıların ihtiyaç duydukları bilgilerin, yazılımlar tarafından daha hızlı bulunması ve hazırlanması mümkün olacaktır. Bilgisayarların bu işlevleri yerine getirebilmesinin önündeki en büyük engel ise, internet üzerindeki mevcut verinin çok büyük bir kısmının yapısal olmayan biçimde, insan doğal dili ile yazılmış olmasıdır. Bu engeli ortadan kaldırmak için, sadece insanlar tarafından anlaşılabilen, doğal dilde yazılmış veriye ek olarak, bilgisayarlarında anlayabileceği, yapısallığı daha yüksek, anlamsal verinin kullanılması önerilmiştir. Bahsedilen bu yeni veri, internete yeni bir üst katman olarak eklenecek ve bilgisayarların anlayabildiği, yorumlayabildiği ve kullanabildiği, semantik ağ oluşturulmuş olacaktır. Semantik veriler, Kaynak Tanımlama Çatısı (Resource Description Framework - RDF), Ağ Ontoloji Dili (Web Ontology Language-OWL) ve Genişletilebilir İşaretleme Dili (Extensible Markup Language-XML) gibi diller kullanılarak yayınlanabilir. Bu verilerin yapısal olarak saklanmasını ve sorgulanmasını sağlamak amacı ile, özelleştirilmiş veritabanı sistemleri (Triple-Store) kullanılmaktadır. Ancak bu çözümlerin

büyük bir kısmı, verilerin ölçeklenebilir olması gereksinimini karşılayamayacak biçimde, tek bilgisayar üzerinde çalışan tasarımlara sahiptirler. Diğer bir grup veritabanı çözümü ise RDF verileri için özelleştirilmemiş olan ilişkisel veritabanlarının kullanımı ile çalışmaktadır. Bu durum, verilerin boyutunun çok büyük olması halinde ve ilişkisel veritabanları çizge verilerini depolamak amacı ile özelleştirilmediği için, sorguların cevaplanmasının çok uzun zaman almasına sebep olmaktadır. Ayrıca üçlü veritabanları ilişkisel veritabanlarının sağlayamadığı ve OWL kullanılarak gerçekleştirilen çıkarım işlemlerini de yapabilmektedir. Bu tez çalışması kapsamında, yukarıda belirtilen problemlerin çözülebilmesi için kullanılabilecek dağıtık bir depolama ve sorgulama altyapısı önerilmiş ve tek bilgisayar üzerinde çalışan bir üçlü veritabanı çözümü ile karşılaştırılması yapılmıştır. Önerilen çözüm programlama ile gerçekleştirilmiş ve bu alanda yaygın kullanılan LUBM Veri Üreticisi ile üretilen veri seti kullanılarak, LUBM test sorguları denenmişlerdir. Tasarlanan sistemin dağıtık yapıda olması, sorgulama işlemlerinin daha küçük veri kümeleri üzerinde ve paralel olarak işletilmesine olanak vermektedir. Bu durumun sorgu cevaplama sürelerine olan etkisi, farklı sayıda birim içeren kümeler ile test edilmiş ve sonuçlar karşılaştırmalı olarak verilmiştir.

**Anahtar Kelimeler:** Semantik ağ, RDF, SPARQL, Dağıtık sorgu işleme, Dağıtık veri depolama, Jena.

| | | |
|---|---|---|
| **University** | : | **TOBB University of Economics and Technology** |
| **Institute** | : | **Institute of Natural and Applied Sciences** |
| **Science Programme** | : | **Computer Engineering** |
| **Supervisor** | : | **Asst. Prof. Erdoğan DOĞDU** |
| **Degree Awarded and Date** | : | **M.Sc. – December 2013** |

**Özgür EROĞLU**

## EFFICIENT STORAGE AND QUERYING OF SEMANTIC DATA ON A DISTRIBUTED SYSTEM

## ABSTRACT

One of the most important features of the Internet is that it lets information to be connected to each other. In this way, regardless of the publisher of it, contents related to each other, is accessible in the form of a network. However, the abundant number and diversity of the data on Internet, makes it difficult for users to quickly access and process the information they need. In the case that, computers can understand and interpret the data, preparation of information by software that users need, would be much faster. The greatest obstacle for computers to perform these functions is that very large portion of the data available on the Internet is non-structural data which was written in human natural language. To eliminate this obstacle, in addition to the usage of data just written in natural language that can be understood by the people, usage of semantic data which can be understood by computers was recommended. Mentioned new data will be added as a new layer to Internet and will enable computers to understand, interpret and use the new semantic network. Semantic data can be published by using languages such as the Resource Description Framework (RDF), Web Ontology Language (OWL) and the Extensible Markup Language (XML). This data is stored and queried in a structural form in customized database systems which are called Triple-Store. However, a large part of these solutions, the design of which are running on a single computer, does not satisfy the need for data to be scalable. Another group of solutions, use relational databases for the purpose of RDF Storage. In this case, when

the amount of data is very large, queries take very long time as RDBMS's are not optimized to store Graph Data. In addition, relational databases can not provide inference capability which is carried out using OWL as it is performed by triple-stores.

In this thesis work, a distributed infrastructure that can be used to store and query semantic data has been proposed and compared with a single Triple-Store running on a single computer. The proposed system solution has been implemented by programming it and tested with LUBM test queries on a set of artificial data which had been generated one of the most commonly used data set generator, called LUBM Data Generator. As the proposed system solution is a distributed one, query processes run in parallel on smaller data sets. The effect of this situation on the time interval for answering queries, have been tested with clusters including different number of units and results are shown in a comparison chart.

**Keywords:** Semantic web, RDF, SPARQL, Distributed query processing, Distributed storage, Jena.

## TEŞEKKÜR

Çalışmalarım boyunca değerli yardım ve katkılarıyla beni yönlendiren hocam Doç. Dr. Erdoğan DOĞDU'ya ve yine kıymetli tecrübelerinden faydalandığım TOBB Ekonomi ve Teknoloji Üniversitesi Bilgisayar Mühendisliği Bölümü öğretim üyelerine teşekkürü bir borç bilirim.

# İÇİNDEKİLER

# ŞEKİLLERİN LİSTESİ

# ÇİZELGELERİN LİSTESİ

# KOD BLOKLARININ LİSTESİ

# ALGORİTMALARIN LİSTESİ

# 1. GİRİŞ

Düşük maliyetli ve yüksek verimli üretim tekniklerinin gelişmesi sonucu, bilgisayar donanımları ucuzlamış ve bunun sonucu olarak da, bilgisayarlar son kullanıcı açısından daha kolay erişilebilir hale gelmiştir. Ayrıca, kişisel kullanıcıların ve kurumların internet erişimlerinin ucuzlaması ve yaygınlaşması, bilgisayar kullanımını gündelik yaşamın vazgeçilmez bir parçası haline getirmiştir. Artan bu kullanım sonucunda hem internet ortamında, hem de diğer alanlarda bulunan veri miktarı ve çeşitliliği çok hızlı bir şekilde artmıştır. Ancak internet üzerindeki bu verinin çok büyük bir kısmı yapısal olmayan (unstructered) nitelikte ve sadece insan kullanıcıların kullanımı öngörülerek hazırlanmış dolayısı ile büyük bir kısmı insan doğal dillerinde oluşturulmuş olan verilerdir. Yapısal olmayan veriler, web sayfalarının içeriğini oluşturan web dokümanları ve web sayfaları içerisinden sunulan diğer doküman türleridir. Yapısal olmayan bu veriler, bilgisayar uygulamaları açısından, etkin şekilde kullanılabilir veriler değillerdir. Bilgisayar uygulamalarının bu veriyi anlamsal olarak işleyebilmesi, kullanıcılar tarafından ihtiyaç duyulan hizmeti daha kolay, daha doğru ve daha zengin bir şekilde sunmalarını sağlayacaktır. Mevcut durumda ise , bilgisayarların veya bilgisayar uygulamalarının, kullanıcının doğrudan yönlendirmesine ihtiyaçları vardır. Bilgisayar uygulamaları, kullanıcının verdiği komutların veya uygulamaya verdiği girdilerin anlamsal karşılıklarının ne olduğunu bilmeksizin işlem yapmaktadırlar. Anlamsal karşılıkların bilinmesi durumunda, uygulamalar, girdinin sadece anlamına göre işlem yapabilecek ve sonuç olarak kullanıcı açısından daha doğru bir sonuç üretebilecektir.

Verilerin, kendilerini anlamsal açıdan ifade eden ek bilgilerle (meta-data) birlikte

üretilmesi ve dağıtılması fikri, internet alanında anlamsal ağ (Semantic Web) kavramının ortaya çıkmasını sağlamıştır. Bu kavram, insanlar tarafından okunabilen ve birbirine bağlı içeriklerin oluşturduğu ağa, anlamsal içeriklerinde dahil edildiği ayrı katmanlar eklenmesini öngörmektedir. Bilgisayar uygulamaları tarafından da anlaşılabilen, yorumlanabilen ve kullanılabilen, yeni bir ağ oluşturulması amaçlanmıştır. Bu yeni ağ, eskisi gibi birbirine bağlı içeriklerin oluşturduğu bir ağ olmasına ek olarak, içeriği sadece insanlar tarafından anlaşılabilen değil, doğrudan bilgisayar uygulamaları tarafından da kullanılabilen bir ağ olacaktır. World Wide Web'i icat eden ve geliştiren Tim Berners-Lee, anlamsal ağ konusundaki düşüncelerini "bilgisayarlar tarafından doğrudan işlenebilen verilerin ağı" ifadesi ile özetlemiştir [14].

## 1.1 Semantik Ağ

Semantik ağın en önemli amacı, mevcut internet ağının evrimleşerek, kullanıcıların ve bilgisayar uygulamalarının bilgiye daha hızlı ulaşmasını, bilgiyi kendi aralarında paylaşmasını ve bilgileri anlamsal olarak birbirine bağlamasını sağlamaktır. Farklı bir ifade ile, semantik ağ, farklı türde ve yapıdaki içeriklerin ve bu içeriklerin bilgisayar yazılımları tarafından anlaşılabilmesini sağlayan üst verilerin bütünleşik halinin adıdır. Mevcut internet ağı ile semantik ağ arasındaki fark, mevcut internet ağının "dokümanların ağı" olarak nitelendirilmesine karşın, semantik ağın "verilerin ağı" olarak nitelendirilmesidir. Semantik ağ ile ilgili kaynaklarda sıkça kullanılan "Linked Data (bağlı veri)" ifadesinin kullanılmasının sebebi budur. Mevcut internet ağı üzerinde oluşan bu yeni anlamsal katman, birbirine bağlanabilen ve farklı sunucular tarafından sunulan verileri içermektedir.

Semantik web fikrinin hayat bulabilmesi için daha önceden ortaya konmuş çeşitli web teknolojileri, W3C (World Wide Web Consortium) çatısı altında standartlaştırılmıştır. Standartların oluşturulması, bu alanda çalışmaların daha hızlı yapılabilmesine olanak vermektedir. W3C tarafından standartlaştırılan en önemli semantik ağ teknolojileri RDF [48], SPARQL [53], OWL [49] ve SKOS'dur [51]. Bu teknolojiler, bağlı verinin belirli bir yapıda yayınlanmasını, sorgulanmasını

ve anlam/sonuç çıkarımı amacı ile kullanılmasını sağlamaktadırlar. Aşağıda, bu teknolojilerden bazıları hakkında kısa bilgiler verilmiştir.

### 1.1.1 RDF ve Çizge Veritabanları

#### 1.1.1.1 RDF

"Kaynak tanımlama çatısı" olarak tanımlanan RDF (Resource Description Framework) [50] semantik ağ konusunun en temel yapı taşlarından birisidir. Bir benzetim ile tanımlama yapmak gerekirse, HTML ve WWW arasındaki bağlantı, RDF ve Semantik Ağ arasında bulunmaktadır. Yani RDF semantik ağın veri yapısıdır. 1999 yılında, W3C tarafından standartlaştırılan, web sayfalarında bulunan bilgiyi tanımlamak için kullanılan üst verilerin (meta data) kodlanması amacı ile üretilmiş bir web teknolojisidir [56]. Üst verilerin bilgisayar uygulamaları tarafından anlaşılır biçimde oluşturulmuş olması, ağ ölçeğinde otomasyon uygulamalarının önündeki problemleri kaldırmış, ayrıca, kullanıcıların kendi çabaları ile kısa sürede işleme imkanı bulamayacakları büyüklükteki bilgilerin, yazılım ajanları tarafından, kullanıcılar adına işlenebilmesinin yolunu açmıştır. RDF, herhangi bir alana özgü veriyi tanımlayabilecek biçimde esnek tasarlanmıştır. Kimyasal maddelerin tanımlanması amacı ile kullanılabileceği gibi, tarihsel olaylara ait tarih, yer gibi verilerin yayınlanması amacı ile de kullanılabilir. Bu, alan bağımsız tasarım kararının en önemli getirisi, yazılımların, bilgisayarlar tarafından anlaşılabilen bilgiyi kendi aralarında paylaşabilmeleri olmuştur. RDF'in diğer bir faydası ise, internet üzerindeki kaynakları tanımlaması dışında bu verilerin aralarındaki ilişkiyi de tanımlaması ve bilgileri birbirine bağlamasıdır.

W3C tarafından yayınlanan RDF tanımlama dokümanında (Resource Description Framework specification ) [48] daha formal bir biçimde yapılan RDF tanımı ve RDF özellikleri aşağıdaki maddeler şeklinde sıralanabilir;

- RDF, WWW (World Wide Web) üzerindeki kaynaklar hakkındaki bilgiyi

ifade eden bir dildir.

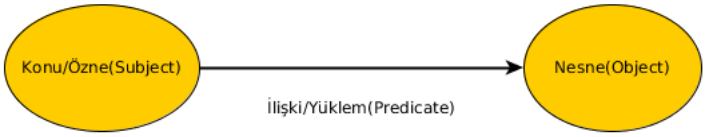
- RDF, İnternet ağı üzerindeki bilgiyi ifade etmek için kullanılabilecek bir uygulama çatısıdır.
- Bilgiyi bilgisayarlar tarafından anlaşılabilecek bir biçimde ifade etmeyi sağlar.
- Özne, nesne, yüklem (subject, object, predicate) başlıklarından oluşan üçlü (triple) yapısını ve bu yapı içinde özne, nesne ve yüklem yapılarının URI kullanılarak tanımlanmasına olanak sağlar.
- RDF her türlü bilginin ifade edilmesini sağlayacak kadar esnek yapıdadır.
- Çizgelerin (graph) ifade edilmesi için bir yol sağlar.
- Özne, nesne, yüklem üçlülerin de URI kullanılması internet üzerinde bulunan dağıtık bilgilerin birbirlerine bağlanmasını sağlar.

RDF'in, bilgileri, hem bilgisayarların anlayabileceği şekilde yapısal olarak ifade etmesini, hem de bu yapı kullanılarak alandan bağımsız olarak her türlü bilginin tarif edilebilmesini RDF soyut modelini (RDF abstract model) sağlamaktadır.

RDF, bu soyut modeli, bilginin anlamsal özellikleri ile ilgili, basit bazı kurallara dayanarak, küçük parçalara ayırmak için kullanmaktadır.Ayırma işleminin amacı bir yandan bilgiyi esnek bir şekilde ifade edebilmeye devam ederken, bir yandan da basit bir yapısallık sağlamaktır. Bu soyut modelin üç adet ana bileşeni bulunmaktadır. Bunlar cümle (statement), özne ve nesne (subject and object) ve yüklemdir (predicate).

İfade edilmeye çalışılan bilginin küçük parçalara ayrılması ile oluşan en küçük anlamsal yapıya cümle denilmektedir. Bir bilgi, bir veya daha fazla cümlenin kullanılması ile ifade edilebilmektedir. Her cümle, aynı zamanda "üçlü" olarak da tanımlanabilen bir yapıya sahiptir. Bu yapı, sırası ile özne, yüklem (veya ilişki) ve nesne bileşenlerinden oluşmaktadır. Dolayısı ile her üçlü, bir özne ve nesneyi, aralarındaki ilişkiye de belirtecek şekilde ifade etmektedir. Daha sonraki

kısımlarda değinilecek olan çizge veritabanlarına kaynak oluşturan düşünce, Şekil 1.1'de verilen üçlü gösteriminden de anlaşılabileceği gibi, her bir cümlenin, küçük bir yönlü çizge olmasıdır.




Şekil 1.1: RDF üçlü gösterimi

Bu yapı kullanılarak oluşturulmuş örnek bir bilgi listesi aşağıda verilmiştir. Örneğin, İngilizce olarak "Ford Mustang manufactured_by Ford Motor Company." şeklinde verilen üçlü, Türkçe'de "Ford Mustang -üretici- Ford Motor Company" şeklinde oluşturulabilir. Bu ifade basitçe, "Ford Mustang, Ford Motor Company tarafından üretilmiştir" cümlesinin, RDF üçlüsü olarak yazılmış biçimidir. Örnekteki RDF üçlüsü içinde özne Ford Mustang, nesne Ford Motor Company ve "üretici" ifadesi ise nesne ve özneyi birbirine bağlayan ilişkidir. Yukarıda verilen örnek genişletilecek olursa, farklı cümlelerden oluşan bir tablo elde edilebilir. Çizelge 1.1'de birden fazla cümle kullanılması ile, hem alan hakkında genel bilgi verilmesi, hem de bu verinin yapısal bir bilgi olması sağlanmıştır.

Çizelge 1.1: Cümle (üçlü) Örnekleri

| Özne | İlişki | Nesne |
|---|---|---|
| Ford Mustang | üretici | Ford Motor Company |
| Ford Mustang | bir (is a) | Araba |
| Ford Mustang | türü | 2 Kapılı |
| Ford Mustang | ilk üretim tarihi | 9 Mart 1964 |
| Ford Motor Company | merkez | Michigan, ABD |
| Ford Motor Company | kurucu | Henry Ford |

Bir üçlü ile sadece bir tek olgu ifade edilebilirken, birden fazla üçlü kullanılarak bir alan hakkında genel bilgi verilebilmektedir. Birden fazla üçlü kullanılması ile

oluşan yapıya bir "RDF çizgesi" denilmektedir.

RDF modeli tarafından kullanılan özne ve nesne, somut objeleri ifade etmek için kullanılabileceği gibi, soyut kavramları ifade etmek amacı ile de kullanılabilirler. Bu somut veya soyut kavramların tamamına "kaynak" denilmektedir. Kaynak Tanımlama Çerçevesi ifadesinde bahsedilen kaynaklar, burada belirtilen somut nesneler veya soyut kavramlardır. Kısaca özne ve nesne olarak daha önce belirtilmiş olan üçlü bileşenleri, kaynak isimleridir.

RDF modelinin sağlamış olduğu diğer bir önemli yetenek ise, tanımlanmaya çalışılan kaynakların, internet üzerinden birbirine bağlanabilmesidir. Bu yetenek sayesinde, kaynak tanımlama sırasında karşılaşılabilecek iki önemli probleme çözüm getirilmiş olur. Bunlar, kaynak tanımlamada kullanılan isimlerinin birden çok anlamlı olması durumu ve bir kaynağın birden fazla isim ile ifade edilebilir olması durumudur. Farklı iki nesneyi tanımlamaya çalışan farklı iki RDF dokümanı, bu iki farklı nesne için aynı ismi kullanmış olabilirler. Benzer şekilde, aynı nesneyi tanımlamaya çalışan iki farklı RDF dokümanı, bu nesne için farklı isimler kullanmış olabilir. Örneğin, "dil" kelimesi kaynak tanımlaması sırasında bir organı ifade edebileceği gibi, bir lisanı ifade etmek amacıyla da kullanılabilir. Bu durum, anlamsal ağ oluşturma çabaları için oldukça önemli bir problem oluşturmaktadır. Dolayısı ile, aynı nesneler aynı isimlerle tanımlanmalı, birden fazla nesnenin tanımlanması için aynı isim kullanılmamalıdır. Bu sorunun çözümü, RDF modelinde, özne ve nesne bileşenlerinin Tek Biçimli Kaynak Tanımlayıcı (URI: Uniform Resource Identifier) olarak kullanılmasıdır. Bu sayede tanımlanacak olan kaynak ile bağlantılı ağ adresleri kullanılarak, daha önceden tanımlanmış kaynaklar RDF dosyasına bağlanabilmektedir. Bu sayede doğru tanımlanmış kaynaklar diğer kaynakların tanımlanması sırasında kullanılabilir olmaktadırlar. Tanımlama amacı ile URI kullanımının en önemli getirisi, kaynak isimlerinin tekillik sağlamasıdır. Bu sayede kaynağın kime ait olduğu, kim tarafından oluşturulduğu konusu netleştirilmiştir. Örneğin, W3C, http://www.w3c.org ile başlayan URI adreslerin sahibidir. Kaynak tanımlamasında, isimler için bu URI'ın kullanılması durumunda, kullanıcın, kaynağın tanımının doğruluğu konusunda bir şüphe duymasına gerek kalmaz.

URI kullanımının diğer bir avantajı ise, nesne veya öznelerin aynı kelimeler ile isimlendirilmesi durumunda dahi, URI'ların aynı olamayacak olması dolayısı ile, çakışmaların olmadığı bir isim uzayı oluşturulmuş olur.

Kaynakları tanımlamak amacı ile kullanılan iki tip URI bulunmaktadır. Bunlar "hash URI" ve "slash URI" tipleridir. Bu iki türe örnek vermek olarak çizelge 1.2 verilmiştir.

Çizelge 1.2: URI türleri

| | |
|---|---|
| http://somedomainaddress/cars/Ford_Mustang | slash URI |
| http://somedomainaddress/cars#Ford_Mustang | hash URI |

Burada verilen birinci cümle bir slash-URI, ikincisi ise hash-URI'dır. Genel olarak kullanılan hash URI'dır. İki tür arasındaki en önemli fark, hash URI ile bir içerik alınabilirken, slash URI sadece kaynağa işaret etmek için kullanılır. Örnekteki iki URI aynı kaynağı, yani Ford_Mustang'i tanımlamaktadır. Dolayısı ile daha önce çizelge 1.1 içinde verilen cümlelerde, nesne veya özne olarak Ford Mustang ifadesi kullanılan yerlerde, yukarıda verilen iki URI'den birisi kullanılabilir. Bu durumda çizelge 1.3'de gösterilen cümleler elde edilir. Böylece, tanımı daha önceden yapılmış bir kaynak, nesne veya özne olarak yeni oluşturulan üçlü içinde kullanılmış olur. URI kullanımı ile kaynak tanımlamaların yapılabilmesi, tekrar kullanım avantajı sağlamakta ve her "anlamsal ağ " kullanıcısının, bütün kaynakları kendi başına tekrar tanımlamasını gereksiz kılmaktadır.

Çizelge 1.3: URI kullanımı ile cümle (üçlü) örnekleri

| **Özne** | **İlişki** | **Nesne** |
|---|---|---|
| http://address/cars#Ford_Mustang | üretici | Ford Motor Company |
| http://address/cars#Ford_Mustang | bir (is a) | Araba |
| http://address/cars#Ford_Mustang | türü | 2 Kapılı |
| http://address/cars#Ford_Mustang | ilk üretim tarihi | 9 Mart 1964 |
| Ford Motor Company | merkez | Michigan, ABD |
| Ford Motor Company | kurucu | Henry Ford |

Çizelge 1.3'de görüldüğü gibi URI ile verilen cümlelerde tekrar eden bölümler

dokümanın okunurluğu açısından bir karmaşa yaratmaktadır. Bu problemi çözmek amacı ile tekrar eden kısımlar birer ön ek (prefix) olarak tanımlanır ve uzun URI'lar bu ön ekler sayesinde XML QName (Qualified Name) olarak isimlendirilen daha kısa bir şekilde ifade edilir. Aşağıda bir ön ek tanımlaması ve bu ön ekin kullanıldığı bir QName gösterilmiştir.

Çizelge 1.4: İsim uzayı örnekleri

| **Ön Ek (prefix)** | **İsim Uzayı(namespace)** |
|---|---|
| rdf: | http://www.w3.org/1999/02/22-rdf-syntax-ns |
| dom: | http://somedomainaddress.com/cars |

Çizelge 1.5: İsim uzayı kullanımı

| | | |
|---|---|---|
| dom:Ford_Mustang | üretici | Ford Motor Company |
| dom:Ford_Mustang | bir (is a) | Araba |

Özne ve nesne bileşenlerinin URI olarak ifade edilebilmesine benzer olarak yüklem yada ilişki bileşenleri de URI olarak ifade edilebilir. Benzer bir yaklaşımla URI kullanımı, yükleminde tekrar kullanılabilir şekilde ifade edilmesini, yüklem olarak kullanılan kelimelerin benzer olması veya eş anlamlı kelimelerin kullanılması durumunda, kastedilen anlamlarının netleşmesini sağlamaktadır. Yüklemin URI olarak ifade edilebilmesinin diğer bir getirisi ise, bu URI'ın başka RDF cümlelerinde özne olarak kullanılabilmesini sağlaması ve bu sayede bahsedilen yüklem hakkında daha fazla bilginin verilebilmesidir.

Nesne isimlendirilmesi sırasında kalıp deyim (literal) ifadelerde kullanılabilir. Bunlar genel olarak üçlünün konu bileşeninin bir özelliğini ifade etmek için verilen değerlerdir. Örneğin, konu bileşeninin ağırlığı, rengi, vb. özelliklerinin ifade edilmesi için kullanılırlar. Kalıp deyim ifadeler sadece nesneler için kullanılabilirler, konu ve yüklem olarak kullanılmazlar.

RDF çizgelerin de, bir URI veya literal değer ile ifade edilmeyen öğelerde bulunabilir. Bu öğelere boş öğe (Blank Node) denir. Boş öğeler sadece özne veya nesne olarak kullanılabilirler.

İnternet ortamında birçok çalışma grubu ve firma mevcut internet verilerinin, RDF formatında yayınlanması projeleri yürütmektedirler. Bunlardan en önemli olanlarından birisi DBPedia projesidir. DBPedia, Wikipedia içerisindeki verilerin RDF formatında yayınlanmasını sağlamaktadırlar. Ayrıca, internet üzerinde bulunan diğer veri kümeleri ile, kendi verilerini bağlayarak uygulamaların DBPedia sorgularından elde ettikleri sonuçların zenginleştirilmesini sağlamaktadırlar.

#### 1.1.1.2 Çizge Veritabanları ve Üçlü Veritabanları (Triple-Store)

Çizge veritabanları özellikle çizge yapısında bulunan verilerin depolanması amacı ile kullanılan özel veritabanlarıdır. Verinin, çizge düğüm öğeleri ve onları ilişkilendiren bağlantı yapıları olarak depolanmasını sağlarlar. Verilerin bağlantılı yapısı sayesinde, bir indeks yapısına gerek yoktur. Her çizge düğümü bağlantılı olduğu komşu düğüm öğelerine bir işaretçi içermektedir. Genel olarak çizge teorisinin sağladığı teorik yapı üzerine kurgulanmış ve uygulamaları yapılmıştır. Ancak benzer uygulamaların, ilişkisel veritabanları kullanılarak yapılması da mümkündür.

Çizge veritabanlarında ilişkisel veritabanlarında olduğu gibi bir şema (schema) ihtiyacı bulunmamaktadır. Bu özellikleri ile verilerin genel amaçlı ve değişken sistemlerde kullanımına daha uygundurlar. Ölçeklenebilirlikleri açısından ilişkisel veritabanlarından daha başarılı sonuçlar vermektedirler. Doğal olarak çizge veritabanları, çizge formunda bulunan verilerin depolanması için daha uygundurlar.

Birçok açık kaynak ve ticari uygulaması bulunan çizge veritabanları, yazıldıkları dil, sağladıkları sorgulama dili desteği, geliştiriciler için sağladıkları programlama arayüzleri ve ölçeklenebilirlikleri açısından farklılıklar gösterebilmektedirler.

RDF verilerinin yönlü birer çizge ifade ediyor olması, bu verileri saklamak amacı ile çizge veritabanlarının kullanılmasını mümkün hale getirmektedir. Özellikle RDF verilerinin saklanması amacı ile kullanılan çizge veritabanlarına üçlü veritabanı (Triple-Store) denilmektedir. İlişkisel veritabanlarının sağladığı kullanım yapılarına paralel olarak, çizge veritabanları da verilerin depolanmasını

ve sorgulanması sağlamaktadırlar. Hemen hemen hepsi sorgulama dili olarak SPARQL sorgu dilini desteklemektedirler. SPARQL sorgularını işleyen ve bunlara göre cevaplar hazırlaya bir SPARQL işleme birimi içerirler. Birçoğu veritabanına yüklenen verinin değiştirilmesine olanak vermesine karşı bazı veritabanları sadece sorgulama işlemine izin vermektedirler.

En yaygın olarak kullanılan, semantik veri işleme uygulama geliştirme arayüzlerinden birisi olan Apache JENA projesi, HTTP, API ve komut satırından sorgu cevaplayabilen Fuseki [11] isimli bir SPARQL uç birimi (end point) içermektedir. Bu yazılım, arka planında yine Jena tarafından sağlanan, Jena-TDB [45] ve Jena-SDB [44] isimli iki farklı üçlü veritabanı içermektedir.

### 1.1.2 RDF Gösterimleri

RDF modellerinin gösterimleri için farklı biçimler bulunmaktadır. Bunların en sık kullanılanı W3C tarafından geliştirilmiş olan RDF çizgelerini XML dosyaları olarak göstermeyi sağlayan, RDF/XML gösterim biçimidir. W3C tarafından geliştirilen başka bir biçim ise Notasyon3 (N3)[1] olarak bilinen gösterim biçimidir. Bunlar dışında en yoğun kullanılan gösterim biçimleri Turtle[2] ve N-Triple[3] gösterim biçimleridir. Hepsi aynı verinin farklı biçimlerde gösterilmesi amacı ile kullanılmaktadırlar. En basit olan gösterim biçimi N-Triples biçimidir.

N-Triples gösterim biçiminde, her cümle bir satır olacak şekilde ve düz metin olarak gösterilir. Her cümle bir SPO (Subject Predicate Object) içermekte bir nokta ile sonlanmaktadır. Özne bir URI veya boş öğe olabilir. Nesneler ise boş öğe veya kalıp deyim olabilir. URI olarak verilen değerler "<>" işaretleri arasında yazılır. Boş öğeler ise önlerinde "_:" olacak şekilde gösterilirler. Aşağıda N-Triple olarak gösterilen bir RDF verisi bulunmaktadır.

[1] http://www.w3.org/TeamSubmission/n3/
[2] http://www.w3.org/TR/2012/WD-turtle-20120710/
[3] http://www.w3.org/2001/sw/RDFCore/ntriples/

N-Triple 1.1: N-Triple yapısında RDF

```
<http://www.w3.org/2001/sw/RDFCore/ntriples/>
    <http://www.w3.org/1999/02/22-rdf-syntax-nstype>
    <http://xmlns.com/foaf/0.1/Document> .
<http://www.w3.org/2001/sw/RDFCore/ntriples/>
    <http://purl.org/dc/terms/title> "N-Triples"@en-US .
<http://www.w3.org/2001/sw/RDFCore/ntriples/>
    <http://xmlns.com/foaf/0.1/maker> \_:art .
<http://www.w3.org/2001/sw/RDFCore/ntriples/>
    <http://xmlns.com/foaf/0.1/maker> \_:dave .
\_:art <http://www.w3.org/1999/02/22-rdf-syntax-nstype>
    <http://xmlns.com/foaf/0.1/Person> .
\_:art <http://xmlns.com/foaf/0.1/name> "Art Barstow".
\_:dave <http://www.w3.org/1999/02/22-rdf-syntax-nstype>
    <http://xmlns.com/foaf/0.1/Person> .
\_:dave <http://xmlns.com/foaf/0.1/name> "Dave Beckett".
```

Aynı verinin RDF-XML olarak gösterimi ise aşağıdaki gibidir.

XML 1.2: RDF-XML yapısında RDF

```

<rdf:RDF xmlns="http://xmlns.com/foaf/0.1/"
         xmlns:dc="http://purl.org/dc/terms/"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns">
   <Document rdf:about="http://www.w3.org/2001/sw/RDFCore/ntriples/">
     <dc:title xml:lang="en-US">N-Triples</dc:title>
     <maker>
       <Person rdf:nodeID="art">
         <name>Art Barstow</name>
       </Person>
     </maker>
     <maker>
       <Person rdf:nodeID="dave">
         <name>Dave Beckett</name>
```

```
15        </Person>
16      </maker>
17    </Document>
18 </rdf:RDF>
```

### 1.1.3 SPARQL

SPARQL, semantik verilerin (RDF çizge) sorgulanması amacı ile kullanılan sorgu dilidir. W3C altında faaliyet gösteren RDF Data Access Working Group (DAWG) tarafından standartlaştırılmıştır. 2013 yılında 1.1 versiyonu yayınlanmıştır [53, 26].

SPARQL'in farklı programlama dillerinde gerçekleştirimi yapılmış ve birçok uygulama programlama arayüzü (API) aracılığı ile geliştirilen uygulamalar içerisine gömülebilirliği sağlanmıştır. Ayrıca SPARQL sorgularının SQL ve XQuery gibi diğer sorgu dillerine çevrilmesini sağlayan uygulamalarda bulunmaktadır.

Genel yapısı, aşağıdaki örnekte verildiği gibidir.

SPARQL 1.3: SELECT sorgu örneği

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name ?email
3 WHERE {
4   ?person a foaf:Person.
5   ?person foaf:name ?name.
6   ?person foaf:mbox ?email.
7 }
```

Yukarıda verilen örnek bir SELECT sorgusuna aittir. 1. satır, sorgu içinde kullanılan ön ek (namespace) tanımını içermektedir. Eğer sorgu içerisinde birden fazla kısaltma kullanılacaksa, her biri sorgunun en üstünde bulunan bu bölüme yeni bir satır olarak eklenmelidir. 2. satır içinde geçen "SELECT", sorgu türünü

ve "?name" ile "?email" ifadeleri, aranan çizge düğümlerini göstermektedir. Aranan çizge düğümleri birer değişkendir ve soru işareti ile başladıkları sürece farklı kelimeler ile kullanılabilirler. 3. ve 7. satırlar arasında kalan kısım, aranan değişkenlerin sağlaması gereken çizge üçlülerini göstermektedir. Sadece, 4.,5. ve 6. satırların hepsini sağlayan değerler cevap olarak dönecektir. Buradan da anlaşılacağı üzere, aslında her üçlü satırı mantıksal bir "VE" işlemi ile birbirlerine bağlıdırlar. Cevap olarak dönen değerler üçlü içindeki SPO bileşenlerinden birisidir. Üçlünün kendisi dönmemektedir.

SELECT sorusu dışında sorgu türleri de bulunmaktadır. Bunlar ASK, CONSTRUCT ve DESCRIBE sorgularıdır. Aşağıda bu sorgu türlerinin amacı hakkında kısa bilgiler verilmiştir.

ASK sorguları, içerdiği üçlüleri sağlayan değişken değerlerinin neler olduğunu değil, sadece RDF içinde var olup olmadıklarını soran sorgulardır. Dolayısı ile cevap olarak "True" veya "False" değerlerini alabilir. Sorguyu cevaplayan birim, sorgudaki üçlüyü RDF içerisinde birden çok değerin sağlaması durumunda dahi,ilk kaydı bulduğu an arama işlemini durdurarak True veya False cevabını gönderir. Aşağıda, Amazon nehrinin Nil nehrinden uzun olup olmadığını soran bir ASK sorgusu örneği bulunmaktadır.

SPARQL 1.4: ASK sorgu örneği

```
1 PREFIX prop: <http://dbpedia.org/property/>
2 ASK
3 {
4   <http://dbpedia.org/resource/Amazon_River> prop:length ?amazon .
5   <http://dbpedia.org/resource/Nile> prop:length ?nile .
6   FILTER(?amazon > ?nile) .
7 }
```

CONSTRUCT sorguları genel olarak SELECT sorguları ile aynı olmakla birlikte sorgulara verilen cevaplar, sorgunun belirttiği üçlüleri sağlayan değerlerin yeniden yapılandırılmasına olanak vermektedir. Yani CONSTRUCT sorgularının sonucu ile elde edilen değerler, sorgu içinde yeni bir çizge oluşturmada kullanılabilir.

SELECT sorgusundan farklı olarak üretilen yeni RDF içinde, üçlü bileşenleri farklı ilişkilerle birbirlerine bağlanabilir.

SPARQL 1.5: CONSTRUCT sorgu örneği

```
CONSTRUCT { ?q1 :hasUncle ?q2 . }
WHERE {
      ?q2     :hasSister ?s .
      ?q1     :hasMother ?s .
}
```

Yukarıdaki örnekte sorgulanan üçlüleri sağlayan ?q1 ve ?q2 değerleri, yeni RDF içinde farklı bir şekilde ilişkilendirilmişlerdir.

DESCRIBE sorguları, doğrudan alt çizgelerin cevap olarak istenmesini sağlayan sorgulardır. Sorgu cevaplayan birim cevap olarak gönderilecek olan çizge içinde hangi üçlülerin bulunacağı hesaplar, çizgeyi oluşturarak cevabı gönderir. Örnek olarak George Washington ile ilgili tüm bilgileri istemek için aşağıdaki sorgu kullanılabilir.

SPARQL 1.6: DESCRIBE sorgu örneği

```
PREFIX prop: <http://dbpedia.org/property/>
DESCRIBE *
WHERE {
      dbpedia:George\_Washington ?anyProperty ?s .
}
```

SPARQL sorguları, sorguları cevaplamak için geliştirilmiş olan ve genel olarak tüm üçlü veritabanlarında bulunan bir birim tarafından cevaplanır. Bu birim SPARQL Uç Birimi (SPARQL Endpoint) olarak adlandırılır. Uç birimler farklı şekillerde sorgulanabilmektedir. Bu yöntemler arasında, HTTP protokolü kullanılarak web üzerinden sorgulama, Uygulama Programlama Arayüzü (API) üzerinden gönderilen sorgulama ve Komut Satırı kullanılarak yapılan sorgulamalar sayılabilir. Birçok üçlü veritabanı, aynı anda bu yöntemlerin birden fazlasını sağlayabilmektedir.

## 1.2 Dağıtık Sistemler ve Ölçeklenebilirlik

Veri işleme süresi ile veri büyüklüğü arasında doğru bir orantı bulunmaktadır. Veri miktarı arttıkça, verinin bir bilgisayar üzerinde işlenebilmesi için gereken süre artmaktadır. Bu süreyi azaltmak amacı ile problemin ölçeğine ve verinin yapısına bağlı olarak farklı yöntemler kullanılmaktadır. Bu tür problemlerde, veriyi işlemede kullanılacak bilgisayarların donanımlarının güçlendirilmesi, bir noktaya kadar çözüm sağlayabilmektedir. Ancak probleme ölçeklenebilir bir çözüm getirebilmek için, çözümün, donanım kaynaklarının gücünden bağımsız olması gerekmektedir. Çünkü, sistem performansı, mümkün olan en gelişkin donanım performansı ile sınırlı olacaktır. Ayrıca, Moore Yasası için atomik seviyede sınırlara yaklaşılıyor olması dolayısı ile, donanım performansına doğrudan dayalı sistem ölçeklenmesinin iyi bir yaklaşım olmadığı öngörülebilmektedir [57]. Donanımların iyileştirilmesi yerine, orta seviye donanımların kümelenmesi ve böylece daha büyük işleme kapasiteleri elde edilmesi yaygın olarak üzerinde çalışılan bir konudur. Bu yaklaşımda genel olarak problemin yapısal olarak özdeş, ancak içerik olarak farklı küçük parçalara bölünmesi, bu parçaların küme bilgisayar düğümlerine dağıtılması ve düğümler üzerinde işlemlerin paralel olarak çalıştırılarak verinin işlenmesi yolu izlenmektedir. Daha sonraki adımda, elde edilen küçük çözüm çıktılarının birleştirilmesi yöntemi ile, problem daha kısa sürede sonuca ulaştırılmaktadır.

İşleme yetenekleri açısından dağıtık bir altyapı kullanılıyor olunması, sistem performansını arttırmaya her zaman yetmeyebilir. Örneğin verinin merkezi olduğu ancak işlem için uç birimlere gönderildikten sonra işlendiği, MPI (Message Passing Interface) benzeri paralel işleme altyapılarında bu durum söz konusudur. Bu yüzden MPI[4] daha çok hesaplamaya yönelik dağıtık bir sistemdir.

Verinin işlem öncesi dağıtık halde uç birimlerde bulunması işlem süresini kısaltacaktır. Ancak bunun sağlanması da tek başına performans artışı sağlamaya yeterli değildir. İşlemlerin verinin depolandığı bilgisayarlarda çalıştırılması ve ara sonuçların bilgisayarlar arasında alış verişinin minimum seviyede yapılması

[4] http://www.mcs.anl.gov/research/projects/mpi/

gerekmektedir. Örneğin GRID[5], dağıtık bir depolama ve işleme altyapısı sunuyor olmasına rağmen, yürütülen işlemler her zaman verinin bulunduğu bilgisayarda çalışmadığı için yoğun bir ağ kullanımı ve dolayısı ile ağ üzerinde daha fazla zaman kaybı anlamına gelmektedir. Dolayısı ile dağıtık bir sistem tasarımında bahsedilen bu noktalar göz önüne alınmalıdır.

Semantik verilerin işlenmesi için sağlanan uygulama geliştirme arayüzlerinin, yukarıda belirtilen yaklaşımlar göz önüne alınarak kullanılması ile benzer uygulamalar geliştirmek mümkündür. Kullanılacak olan çözüm probleme göre belirlenebilir. Veri yapısı açısından özdeş verilerin topluca işlenmeleri amacıyla sıklıkla kullanılan açık kaynak bir altyapı olan Apache Hadoop [6] projesi çok önemli bir örnek teşkil etmektedir.

## 1.3 Problem Tanımı

Mevcut üçlü veritabanlarının en büyük eksikliği ölçeklenebilir olmamalarıdır. Bu durum, test amaçlı çalışmalarda sorun oluşturmamakla birlikte, verinin büyük olduğu ticari ve akademik uygulamalarda büyük bir eksiklik oluşturmaktadır. Verinin birden fazla uç birim üzerinde ayrı ayrı sunulması durumunda ise veri bütünlüğü kaybedilmektedir. Bir sorgunun cevaplanabilmesi için sağlaması gereken üçlü yollarından bir kısmının diğer sunucuda sunuluyor olması, ilgili sunucunun sorguya cevap verememesine sebep olmaktadır. Yani semantik veri, ilişkili olan verilerin birlikte sunulduğu sistemlerde doğru olarak sorgulanabilmektedir. Bu yeteneği dağıtık olarak sağlamak için, fiziki olarak ayrı donanımlar üzerinde bulunan ilişkili verilerin, bu ilişkiler göz önünde tutularak sorgulanabilmesi gerekmektedir.

Verinin parçalanması ve ayrı sunuculardan sunulması durumunda, her sunucu kendi üzerinde bulunan veri ile ilgili kısımlara cevap verebilmektedir. Bu durumda, kullanıcıların sorgularını verinin parçalanışına uygun olarak bölüp,

[5] http://en.wikipedia.org/wiki/Grid_computing
[6] http://hadoop.apache.org

sorgu yapması ve geri gönderilen sorgu sonuçlarını birleştirmek sureti ile istenilen cevaba ulaşması gerekmektedir. Bu yaklaşım iki nedenden dolayı, bir yazılım kullanmaksızın kolay uygulanabilir bir yaklaşım değildir. Birincisi, kullanıcı her sunucuda verinin hangi kısmının olduğunu bilmelidir. İkincisi ise, verinin çok büyük olması durumunda, geri gönderilen sonuçların kısa bir sürede birleştirilmesi gerekecektir. Ancak bu yöntem bir yazılım aracılığı ile yapıldığında çözüm olarak kullanılabilir. Bu durumda dahi, birinci neden olarak belirtilen durumun kısa bir çözümü bulunmamaktadır. Bu durum, kendi içinde iki ayrı alt yöntem ile çözülebilir. Birinci alt çözümde, veri parçalama işlemlerinin her biri, birer veri indeksleme işlemi ile devam eder, böylece, her veri bloğunun içine hangi verilerin olduğu bilinebilir. İkinci alt yöntemde ise, veri bloklarının içeriği bilinmeksizin sorgu parçaları her veri bloğuna gönderilir. Her iki durumda da sorguları bölmek ve cevapları birleştirmek zorunluluğu vardır. İndeks kullanımını gerektiren yaklaşımlarda, tabloların büyüklüğü, indeks içinde arama sürelerini de uzatacaktır.

En yaygın olarak kullanılmakta olan çözüm, ölçeklenebilirlik özelliği bulunan ilişkisel veritabanlarının kullanılması yöntemi ile verinin dağıtık olarak saklanmasını sağlamaktır. Ancak bu çözümde, sorguları alacak ve işleyecek daha sonra ilgili SQL sorguları haline çevirecek bir birimin geliştirilmesi gerekmektedir. Benzer şekilde ilişkisel veritabanından dönen cevapların birleştirilmesi işlemininde bu birim tarafından yapılması gerekmektedir.

Bu tez çalışması, yukarıda anlatılan problemleri çözmek amacı ile kullanılabilecek olan, yazılım sisteminin, çalışma metodunun geliştirilmesini içermektedir.

## 1.4 Tezin Amacı ve Katkıları

W3C tarafından önerilen Semantik Ağ teknolojileri ve bunların standartlaştırılması, bu alanda yapılan çalışmaların hız kazanmasına yardım etmiştir. Ancak Semantik Ağ teknolojilerinin standartlaştırılmaları 1990'lı yılların sonuna doğru olmasına rağmen, semantik ağın en temel bileşeni olan verinin üretimi, son

yıllara kadar aynı hızda gerçekleşmemiştir. Veri üretiminin hızlanması ve artı olarak internet üzerindeki kaynakların semantik veriler haline dönüştürülmesinin hızlanması ile birlikte, verilerin depolanması ve sorgulanması konusu önem kazanmıştır. RDF verilerin, tek bilgisayar üzerinde etkin bir biçimde depolanması ve sorgulanması konusunda bir çok önemli akademik çalışmalar yapılmıştır [2, 36, 29, 18, 22, 34]. Önerilen bu sistemler tek bilgisayar üzerinde performanslı bir şekilde çalışan üçlü veritabanlarıdır. Ancak, semantik veri, yapısı itibari ile çok büyük boyutlara ulaşabilen bir çizge yapısındadır. Dolayısı ile, verinin büyüklüğü sorgulama süresi açısından önemli bir engel oluşturmaktadır. Bu nedenle, semantik ağın geliştirilmesi açısından, verinin ölçeklenebilir dağıtık sistemler üzerinde depolanabilmesi ve hızlı sorgulanabilmesi en öncelikli hedeflerden birisini oluşturmaktadır.

Mevcut üçlü veritabanlarının birçoğu ölçeklenebilirlik yeteneğine sahip olmadıkları için, kısıtlı miktarda veri depolayabilmektedir. Depolanabilen verinin, tek sunucu tarafından servis edilebilmesi durumunda ise, sorgu cevaplama süreleri çok uzun olmaktadır. Dağıtık olarak veri depolayabilen ve sorgulanmasını sağlayan açık kaynak kodlu ve ücretsiz dağıtılan çok fazla üçlü veritabanı bulunmamaktadır [1, 35]. Ancak bu yeteneğe sahip bazı ticari ürünler bulunmaktadır [40, 37].

Yukarıda bahsedilen nedenlerden ötürü, bu tez çalışmasında, semantik verinin, etkili bir biçimde dağıtık olarak depolanabildiği ve sorgulanabildiği bir çözüm oluşturmak hedeflenmiştir. Çalışma sırasında geliştirilecek olan çözüm yaklaşımının, bir uygulama ile gerçeklenmesi tez çalışmasındaki diğer hedefler arasında bulunmaktadır. Uygulama sonucunda oluşturulacak olan sistem, geliştirilen yaklaşımın test edilmesi amacı ile kullanılacak ve sonuçlar tek sunuculu bir sistem ile karşılaştırmalı olarak verilecektir. Geliştirilecek olan tasarımın, düşük ağ kullanım yoğunluğu sağlaması, ölçeklenebilir olması ve sorguları hızlı cevaplaması göz önünde tutulacak tasarım kısıtları olacaktır. Sorguların doğru cevaplar üretiyor olması, ilk etapta sistem performansından daha önemli olacaktır. Bu hedef gerçekleştirildikten sonra, cevaplama sürelerinin düşürülmesi sağlanacaktır.

İlk etapta, veri yedekliliği ve tüm SPARQL yeteneklerinin gerçekleştirilmesi gibi

hedefler göz ardı edilecek, sistemin sadece LUBM sorgu kümesini cevaplayabilir olması sağlanacaktır.

Tezin, bu alanda yapılan çalışmalar açısından getireceği en büyük katkı, mevcut tek sunuculu sistemlere oranla daha kısa sürede sorgu cevaplayabilen ve ölçeklenebilen bir dağıtık sistem önerisi getirebilmesi, ayrıca, bu sistemin, mevcut semantik ağ teknolojilerinin ve açık kaynak kütüphanelerin kullanılması yolu ile oluşturulabileceğinin gösterilmesidir. Getirilen öneri yapılmış olan benzer çalışmalar içinde bir alternatif oluşturacaktır.

# 2. İLGİLİ ÇALIŞMALAR

Tez çalışması sırasında üzerinde durulacak olan temel konu SPARQL sorgularının dağıtık bir ortamda depolanmış olan veriler üzerinde, dağıtık olarak çalıştırılması konusudur. Konuyla ilgili yapılmış farklı akademik çalışmalar bulunmaktadır. Bu bölümde, yapılan bu çalışmaların bazıları hakkında bilgiler verilecek ve kısa değerlendirmeler yapılacaktır. Dağıtık sorgulama ve depolama konusundaki yaklaşım biçimleri ve yöntemleri, bu tez çalışmasının yaklaşımı ile karşılaştırılacaktır.

Dağıtık depolama ve sorgulama alanında yapılan çalışmaların bir kısmı, sıfırdan yeni bir dağıtık üçlü veritabanı geliştirmek yerine, mevcut üçlü veritabanlarının dağıtık olarak kullanılmasını sağlayacak olan, altyapıları geliştirmek yolunu seçmişlerdir [28, 39, 23]. Burada asıl hedef, dağıtık halde çalışması amacı ile tasarlanmamış olan üçlü veritabanlarının, bir yazılım bileşeni olarak kullanılması yolu ile, dağıtık çalışma yeteneği olan bir sistemin veri depolama ve sorgulama katmanını oluşturmasını sağlamaktır. Bu yaklaşımın en önemli nedeni, geliştirme sürelerini kısa tutmak ve halihazırda, geliştirilmiş ve kullanılmış olmaları nedeni ile, yeterli olgunluğa erişmiş yazılımları bileşen olarak kullanmaktır. Ayrıca her üçlü veritabanı, gelen SPARQL sorgularını cevaplamak amacı ile kullanılacak olan sorgu uç birimlerini (SPARQL Endpoint) içermekte olduğundan bu uç birimlerin geliştirilmesi gerekmemektedir. Ancak, bu yaklaşıma alternatif olacak şekilde farklı depolama araçları kullanılması durumunda, depolama amaçlı kullanılan her uç birimde, SPARQL sorguların yürütülmesini sağlamak amacı ile, birer SPARQL uç biriminin geliştirilmesi gerekecektir. Bu yaklaşımın artı yanları olduğu gibi, eksi yanları da bulunmaktadır. Var olan üçlü veritabanlarını kullanmanın getirdiği en önemli eksi ise, mevcut olan üçlü veritabanlarının birçoğunun üçüncü

parti yazılımlarla entegre edilmelerinin kolay olmayışıdır. Entegrasyonun sorun olmasının en önemli nedeni ise, bu üçlü veritabanlarının büyük bir çoğunluğunun bir API ve uzaktan sorgulama için bir erişim protokolü sağlamıyor olmalarıdır.

Üçlü veritabanları, en genel seviyede, gelen SPARQL sorgularını işlemek ile sorumlu olan bir sorgu motoru (Query Engine) ve verinin depolanması için bir veritabanı sisteminden oluşmaktadırlar. Dağıtık bir üçlü veritabanının geliştirilmesi sırasında bu iki bileşenin de, dağıtık yapı içinde bulunması gerekir. Üçlü veritabanları RDF dosyalarının sorgulanması amacı ile optimize edilmiş indeksler oluşturmaktadır. Bu açıdan, mevcut ilişkisel veritabanlarından daha fazla performans göstermektedirler. Tüm üçlü veritabanları, bu işlemleri yerine getiren birer yazılım bileşenini halihazırda çalıştırmaktadır. Hazır bir üçlü veritabanını kullanmak bu bileşenleri geliştirme yükünü ortadan kaldırmaktadır. Ancak, bu iki yapıyı ayrı ayrı geliştirmek ve ayrı ayrı kullanmak mümkündür. Bazı çalışmalar, bu iki bileşenden biri olan veritabanı bileşeni ihtiyacını, mevcut ilişkisel veritabanlarını (RDBMS) veya NoSQL veritabanlarını kullanarak sağlamakta ve SPARQL sorgularını işlemek için mevcut ve genel kullanıma açık projeleri kendi projelerine eklemektedirler [17, 3, 55, 32, 20, 4, 15]. Ancak genel olarak, ilişkisel veritabanları çizge yapılı veri depolama amacı ile optimize edilmedikleri için düşük performans göstermektedirler [3]. Bu çalışmaların bazılarında, sorguların cevaplanması amacı ile SPARQL uç birimleri çalıştırmak yerine, SPARQL sorgularının SQL sorgularına çevrilmesi yolu önerilmiştir [15, 3]. Diğerlerinde ise, kullanılacak olan veritabanının önünde çalıştırılması öngörülen, ayrı SPARQL sorgu uç birimlerinin kullanılmasına olanak veren ara yazılım katmanlarının kullanılmasını önermektedir. En yaygın olarak kullanılan SPARQL sorgu motorlarının bazıları "Sesame2" [38], "ARQ" [12] ve "OpenLink Virtuoso" [37] olarak sayılabilir. Mevcut üçlü veritabanlarının ve SPARQL uç birimlerinin daha geniş bir listesi, W3C web sayfalarında bulunabilir [52]. Veritabanlarının ilişkisel veritabanlarından veya mevcut NoSQL veritabanlarından seçilmelerinin nedenleri farklıdır. İlişkisel veritabanlarının kullanılmasının nedeni, genel olarak, mevcut teknolojik altyapının kullanılabilirliğini, semantik ağ alanına da genişletmek iken, NoSQL veritabanları yüksek ölçeklenebilirlikleri nedeni ile tercih edilmektedirler.

Birçok NoSQL veritabanı bulunsa da, Apache Hadoop temelleri üzerinde geliştirilmiş olan Apache HBase [9] en başarılı ölçeklenebilir NoSQL veritabanlarından bir tanesidir. Temelinde, sadece bir anahtar-değer (key-value) tablosu sunucusudur ve Google tarafından yayınlanan Google BigTable [19] çalışmasını örnek alarak geliştirilmiştir. BigTable, ölçeklenebilirliği sağlayabilmek için ilişkisel veritabanlarının sağladığı bazı OLTP (Online Transaction Processing) özelliklerini uygulamamış veya daha basit olacak şekilde uygulamıştır. Ayrıca yapısal veriler için tasarlanan bu sistemde SQL desteği de bulunmamakta, sorgulama amacı ile SQL benzeri GQL[1] isimli bir dil kullanılmaktadır. Jena-HBase isimli çalışma [32], Hadoop [7] temelli olması dolayısı ile, yüksek hata toleransı, çok etkin bir veri yedekliliği altyapısı ve güçlü bir yük dağılımı yönetimi sunan, HBase veritabanını kullanmaktadır. HBase ile veri sorgulama amacı için, SQL benzeri bir sorgulama dili desteği bulunan, Apache Hive[2] arayüzü kullanılabilmektedir. Ancak API seviyesinde erişim ve sorgulama da mümkündür. HBase, küme büyüklüğüne doğrudan bağımlı bir performans yapısına sahiptir. Bu yüzden, Jena-HBase çalışmasında, veri depolama katmanı olarak HBase tercih edilmiştir. Ancak, HBase performans konusunda başarılı olmasına rağmen, indeksleme açısından sınırlı bir yeteneğe sahiptir [9]. Jena-HBase projesi, sorguların yönetilmesi amacı için Apache Jena uygulama geliştirme arayüzünü (Jena API) [34, 10] kullanır. Bu uygulama geliştirme arayüzü, SPARQL sorgularını işlemek için gerekli birçok fonksiyon sunmaktadır. Jena dağıtımları TDB [45] ve SDB [44] isimli iki üçlü veritabanı ile birlikte kullanılabilmektedir. Bunlar Jena projesi altında geliştirilen üçlü veritabanlarıdır. Ancak Jena sadece bu iki üçlü veritabanı ile sınırlı şekilde çalışmamaktadır. Sağlanan arayüzler sayesinde, gerekli sürücü yazılımları ile birlikte, arka planda çalışacak üçlü veritabanı olarak, kullanıcının seçmiş olduğu başka bir veritabanı ile de çalışabilmektedir. Jena-HBase projesi, Jena uygulama geliştirme arayüzünü kullanarak yazılan SPARQL işleme yazılımın HBase ile kullanılmasını sağlayan sürücü yazılımlarını da geliştirmiştir.

Hadoop temelli benzer başka bir proje olan SHARD [42] ise, verileri, HBase

[1] https://developers.google.com/datastore/docs/apis/gql/gql_reference
[2] http://hive.apache.org/

gibi bir NoSQL veritabanı içindeki tablo objeleri içinde depolamak yerine, Hadoop tarafından sağlanan HDFS [8] dosya sistemini kullanarak, düz metin dosyaları içinde depolama yapmaktadır. Bu durumda, veri okuma yazma işlemleri için doğrudan dosya sistemine erişilebildiği için, zaman kazançları daha fazla olabilmektedir. SHARD, sorgu işleme amacı ile map-reduce uygulamaları kullanmakta ve girilen SPARQL sorgularını, alt sorgular halinde ele alarak, her alt sorgu için verinin tümü üzerinde bir döngü yapmaktadır. Bu döngülerin her birinde, alt sorgu içinde verilen yolu sağlayan değerler, sonuç kümesine eklenmektedir. Bu çözüm yaklaşımı, esasen çok basit ve sadece, Hadoop'un tekrarlı işleri yapmak konusunda çok başarılı ve yüksek zaman kazancı sağlıyor olmasına dayanan bir çözüm yaklaşımıdır. Bu projedeki yaklaşımın gerçekleştirilebilmesi için, SPARQL sorgularının bileşenlerine ayrıldıktan sonra, Hadoop dosya sistemi yöneticisinin bu dosyalara erişimini sağlayan ara yazılım katmanlarının yazılması gerekmektedir. Çünkü dosya bloklarının hangi küme birimleri üzerinde depolandığı sadece Hadoop Namenode sunucusu tarafından bilinmektedir. Veri parçalama işlemi için herhangi özel bir yaklaşım sağlamamaktadır. Bu iş tamamen Hadoop NameNode sunucusuna bırakılmıştır. NameNode, verinin erişilebilirliğini ve yedekliliğini arttırmak amacı ile, kendisine verilen tüm dosyaları, 64 Byte büyüklüğünde bloklar halinde, en az 3 küme birimi üzerine bulunacak biçimde dağıtmaktadır. Bu 3 küme birimi genellikle rastgele seçilmektedir. Dağıtılan bloklar ile ilgili tüm bilgiler, Namenode sunucusunda üst veri(metadata) olarak depolanmaktadır. Bu bilgiler, bloğun hangi küme birimlerinde bulunduğunu gösteren bilgilerdir, verinin içeriği hakkında bir bilgi içermemektedir. Veriye erişim gerektiği zaman, Namenode, küme birimlerinin yük durumlarını göz önüne alarak, hangi blokların hangi küme birimlerinden okunması gerektiği bilgisini sağlamaktadır. Bu açıdan SHARD içinde, verinin nasıl dağıtılacağı Hadoop NameNode tarafından belirlenmektedir.

Verinin parçalanması biçiminin, daha ön planda olduğu, ancak yine Hadoop temelli olan başka bir dağıtık sistem önerisi de Jiewen Huang, Daniel J. Abadi ve Kun Renve tarafından yapılmıştır [28]. Bu çalışmada verinin parçalanma biçiminin, sorgu sürelerine etkileri üzerinde durulmakta ve "directed n-hop guarantee" isimli bir veri dağıtma algoritması ve bunu göz önüne alarak sorgulama

yapan bir sistem önerilmektedir. Algoritma, SPARQL sorgularının yapısına istatistiksel bir yaklaşım ile bakarak, herhangi bir sorgu içinde, sağlanması gereken alt çizge büyüklüğünün belli sınırları aşmadığı tespitini kullanmaktadır. Çizge parçalama amacı ile METIS [27] kullanılmıştır[3]. Bu algoritmaya göre dağıtılan veriler, farklı uç birimlerde tekrarlanmış olarak bulunabilmekte ve depolama etkinliğini azaltmaktadır[4]. Ancak bu durum, birbiri ile alakalı üçlülerin, tek birim üzerinde bulunma ve dolayısı ile bunları sorgulayan sorguların tek birim tarafından cevaplanması şansını arttırmaktadır. Sonuç olarak, uç birimler arası bilgi aktarımları en alt seviyede tutulmuş ve uç birim cevapları ile oluşan veri birleştirme zorunluğu ortadan kaldırılmış olmaktadır. Bu sistem Hadoop altyapısını sorguların cevaplarının birleştirilmesi gerektiği durumlarda kullanmaktadır. Aldığı sonuçlar itibari ile hızlı bir sistem olmasına rağmen, sorgu biçimleri ile ilgili yapmakta olduğu varsayım itibarı ile her sorguda benzer sonuçlar elde etmesi mümkün değildir.

Jena-HBase projesine büyük oranda benzeyen, ancak HBase yerine, Google BigTable benzeri bir anahtar-değer veritabanı kullanan başka bir proje ise Rya [41] projesidir. HBase yerine Accumulo [5] isimli başka bir anahtar-değer tablo sunucusu kullanılmıştır. Accumulo, HBase ile benzer şekilde, Google BigTable tasarımını referans olarak kullanmıştır. Ancak, HBase'den farklı olarak, HBase üzerinde bulunmayan, hesaplama performansı ve güvenlik sağlayan bazı ekstra özelliklere sahip olduğu için Rya projesinde bu veritabanını seçmiştir. Rya projesinin SPARQL işleme katmanında, Jena yerine, benzer yetenekler sağlayan başka bir uygulama geliştirme arayüzü olan Sesame Uygulama Çatısı (Sesame Framework) [38] kullanılmıştır[5]. Jena-HBase projesindekine benzer bir şekilde, sorgular ilk adımda parçalara ayrılmakta ve daha sonra Sesame içinde bulunan, SAIL uygulama geliştirme arayüzü kullanılarak yazılan yazılım birimi tarafından parçalara ayrılmakta, daha sonra veritabanı içinde ilgili kayıtları arama işlemi

[3]Çalışma detaylarının anlatıldığı makalede, diğer bir veri parçalama yöntemi olarak, Hash Partitioning ile veri parçalanması işlemi anlatılmış ve METIS kullanılarak yapılan parçalama işlemi ile kıyaslanmıştır.

[4]Önerilen sistem, depolama alanı kullanma etkinliği ve sorgu cevaplama süreleri arasında bir alakaya dayanmaktadır.

[5]Jena ve Sesame Java programlama dili ile gerçekleştirilmişlerdir, uygulama geliştirme arayüzleri (API) Java için yazılmıştır.

yürütülmektedir. Bu arama işlemi, Accumulo içinde sağlanan Batch Scanner isimli bir birim tarafından yapılmaktadır. Çalışma ile ilgili makalede [41], Accumulo ile HBase arasında en önemli performans farklarından birini, bu birimin sağladığı belirtilmiştir. Çalışma ekibi, elde ettikleri sonuçlarını SHARD ve yukarıda anlatılan çizge parçalama (Graph Partitioning) çalışmaları ile karşılaştırmış, veri yükleme ve sorgu cevaplama süreleri açısından Rya projesinin daha iyi sonuçlar elde ettiğini göstermiştir.

Yukarıda belirtilenler dışında, Hadoop teknolojisine dayanmaksızın, dağıtık RDF üçlü veritabanı geliştiren çalışmalar da bulunmaktadır. Bunlardan en önemlisi 4-Store [1, 23] projesidir. 4-Store projesi üçlüleri model-özne-ilişki-nesne şeklindeki dörtlüler yapısı ile saklamaktadır. Bu projede, kullanılan fiziksel küme yapısı, proje kapsamında geliştirilen TCP/IP tabanlı, birimler arası haberleşme altyapısı üzerinden veri alış verişi yapmaktadır. Küme içinde bulunan bilgisayarlar depolama birimi ve işleme birimi olmak üzere iki sınıfa ayrılmışlardır. İşleme birimleri, sadece depolama birimleri ile haberleşmekte ancak kendi aralarında veri alış verişi yapmamaktadırlar. Depolama birimlerinin her biri verinin belli bir dilimini içermektedirler. Sorgular proje kapsamında geliştirilen SPARQL işleme birimi ile işlendikten sonra işleme birimleri üzerinden çalıştırılırlar. Her işleme birimi, yük dağılımlarını da göz önüne alarak sorguyu cevaplayacak olan depolama birimini tespit ederek sorgu cevabını ister. Verinin dağıtılması basit bir Hashing metoduna dayanmakta olduğu için işlemci birimler cevapların hangi birimlerde olduğunu hesaplayabilmektedirler. 4-Store, diğer alternatiflere oranla, daha uzun süre gerçek kullanım alanında test edilmiş bir uygulama olduğu, hataları en çok ayıklanmış üçlü veritabanlarından bir tanesidir. 4-Store tasarımı, 32 birimlik bir küme için optimize edilmiştir. Ancak daha büyük kümelerde çalıştırılması sonucu elde edilen verilere dair bir bilgi bulunamamıştır[6].

Bu tez çalışmasında önerilecek olan sistem yukarıda kısaca değinilen sistem önerileri ile bazı paralellikler içeriyor olsa da, yakın seviyede benzeyen bir sistem yaklaşımı bulunmamaktadır. Bu tez çalışmasında verinin uç birimlerine

[6] 4-Store üçlü veritabanının Hadoop temelli sistemlerle karşılaştırılmasını içeren bir kaynak bulunamamıştır

dağıtılması sırasında Çizge Parçalama projesinde [28] kullanılan METIS kullanılacaktır. METIS veriyi çizge özelliklerini göz önüne alarak, ilişkili çizge düğümleri aynı çizge parçası içinde kalacak şekilde parçalamakta, bu sayede sorguların mümkün olduğunca az uç birim tarafından cevaplanabilir kalmasını sağlamaktadır. Çizge Parçalama projesinde cevap veren uç birim sayısının daha da düşürülmesi amacı ile bir çizge parçasına atanmış üçlüler, çizge sınırında komşu olunan diğer çizge parçalarına da kopyalanmıştır. Ancak bu tez çalışmasında, Çizge Parçalama projesinde olduğu gibi aynı üçlülerin birden fazla uç birim üzerinde tekrarlanması sağlanmayacaktır. METIS kullanarak, her ne kadar cevap veren uç birim sayısı en düşük seviyeye indiriliyor olsa da, cevabın tamamının tek birimden cevaplanmasını sağlamak amacı ile veri tekrarlama gereği görülmemektedir. Çünkü veri tekrarı yapılması, sistemin toplamda, depolama açısından etkinliği azaltmaktadır. Yukarıda anlatılan sistemlerin çoğunda Hadoop verinin dağıtılması ve sorguların işletilmesi süreçlerine aktif olarak katılmakta olmasına karşın, bizim çalışmamızda, verinin ön işleme süreci dışında Hadoop kullanılmayacaktır. Sorgu işleme için Jena-HBase projesindeki gibi, Jena uygulama geliştirme arayüzü kullanılacaktır. Sesame2 yerine Jena'nın seçilmesinin sebebi, Jena TDB üçlü veritabanının veri yükleme süreçlerinde daha hızlı cevap vermesi ve ölçeklenebilirliğidir [16]. Buna karşın Sesame2 sorgu cevaplama performansları açısından daha başarılıdır. Ancak yapılan deneyler sırasında büyük miktarlarda verilerin çok sık yüklenmesi ve silinmesi gerektiği için Jena seçilmiştir. Diğer bir seçim nedeni, Jena'nın HTTP protokolü üzerinden sorgular cevaplayabilen, Fuseki (SPARQL query end point)[7] isimli alt bileşeninin olmasıdır. Depolama altyapısı olarak ise diğer projelerde kullanılmamış olan Jena TDB üçlü veritabanı seçilmiştir. Bu seçimin sebebi ise, TDB üçlü veritabanının, kullanılan Jena yazılım geliştirme arayüzü ile daha kolay entegre edilebilir olmasıdır. Önerilen sistemde, birimlerin kendi aralarındaki veri alışverişleri için ise Apache Java Commons API [6] kullanılarak HTTP üzerinden haberleşme yapılmasına karar verilmiştir. Sistem içindeki uç birimlerin topolojik yapısı ise, sorguların cevaplanması sürecinde, uç birimlerin birbirlerine sorgu göndermesini sağlayan bir örgü yapısıdır.

[7] http://jena.apache.org/documentation/serving_data/

Burada açıklanan örnek çalışmalara ait özet bilgiler Çizelge 2.1'de verilmiştir.

Çizelge 2.1: İlişkili çalışmaların karşılaştırmalı özeti

| Çalışma Adı | Veri Depolama Altyapısı | Sorgu Katmanı | Veri Parçalama Yöntemi |
|---|---|---|---|
| Vertical Partitioning [3] | PostgreSQL | Sesame | Vertical Partitioning |
| Jena-HBase [32] | HBase | Jena+HBase API | HBase(Hadoop Namenode) |
| SHARD [42] | Hadoop HDFS | Hadoop+MapReduce | Hadoop Namenode |
| GraphPartitioning [28] | RDF3X | RDF3X | Metis |
| Rya [41] | Accumulo | Sesame2 | Accumulo |
| 4Store [1, 23] | 4Store | 4Store | 4Store |

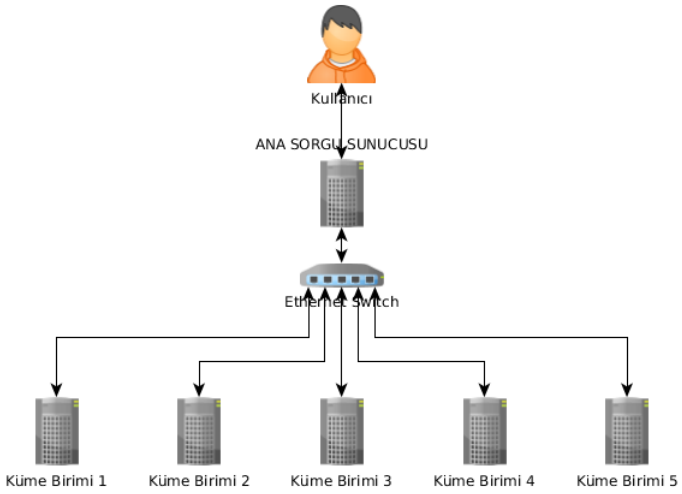
# 3. RDF VERİLERİNİN DAĞITIK ORTAMDA DEPOLANMASI VE SORGULANMASI

Bu bölümde tez çalışması sırasında tasarlanan sisteme ait detaylı bilgiler verilecektir. İlk olarak sisteme genel bir bakış yapılacak, sistemin mantıksal ve fiziksel mimari yapısı hakkında bilgiler verilecektir. Daha sonraki bölümlerde, sırası ile verinin sisteme verilmesi öncesinde yapılan ön işlemler, tezde önerilen çözüm yaklaşımının bir gerçekleştirimi olan DRS (Distributed RDF Store) uygulaması aracılığı ile, verinin parçalanması ve dağıtık ortamda depolanmak üzere ağa gönderilmesi işlemleri anlatılacaktır. Son kısımda sistem üzerine dağıtılmış olan verinin nasıl sorgulandığı açıklanacaktır.

## 3.1 Sistemin Mimari Yapısı

Tez çalışmasındaki en öncelikli hedeflerden birisinin, verinin dağıtık bir sistem üzerinden depolanması ve sorgulanması olması dolayısı ile kullanılan fiziksel sistem bir küme bilgisayar altyapısıdır. Tüm bilgisayarlar birbirlerine Fast Ethernet Anahtarı ile bağlanmışlardır. Sistemin fiziksel yapısı Şekil 3.1 ile gösterilmiştir.

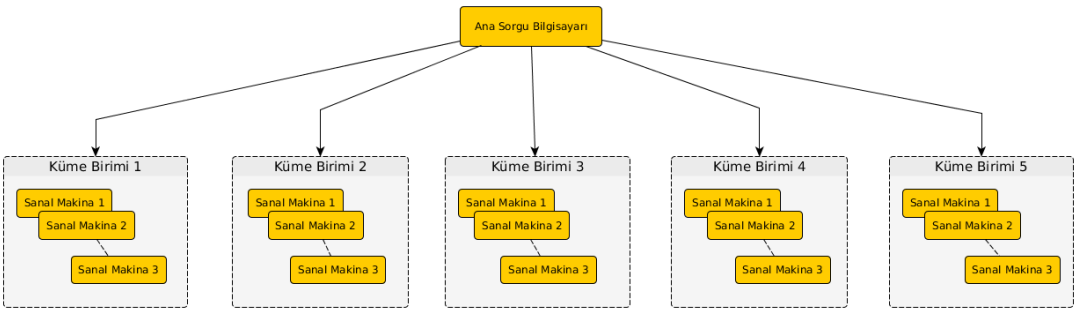
Sistemde bulunan bilgisayarlar fiziksel olarak özdeş olmamakla birlikte, her




Şekil 3.1: Sistemin Fiziksel Yapısı

bilgisayar sanallaştırma sağlayabilen donanımlara sahiptirler. Sistem birimlerinin özdeş olmamasının etkisini azaltmak için, tasarımın uygulanmasında mantıksal seviyede sanal makineler kullanılmıştır. Bu sayede tüm sanal birimlerin özdeş kaynak miktarına sahip olması sağlanmaya çalışılmıştır. Ancak geliştirilen sistemin doğrudan fiziksel bilgisayarlar üzerinde çalıştırılması durumunda dahi, tasarımın çalışmasını engelleyecek bir durumu söz konusu olmayacak, sadece sistem performansı gözlenirken bu durumun bir etkisi olacaktır. Sistem, en yavaş bilgisayarın kendisine verilen iş parçasını bitirmesini beklemek zorunda kalacaktır.

Yukarıda belirtilenler doğrultusunda, tasarımın uygulamasında kullanılan küme, mantıksal seviyede Şekil 3.2'deki gibi görünmektedir.



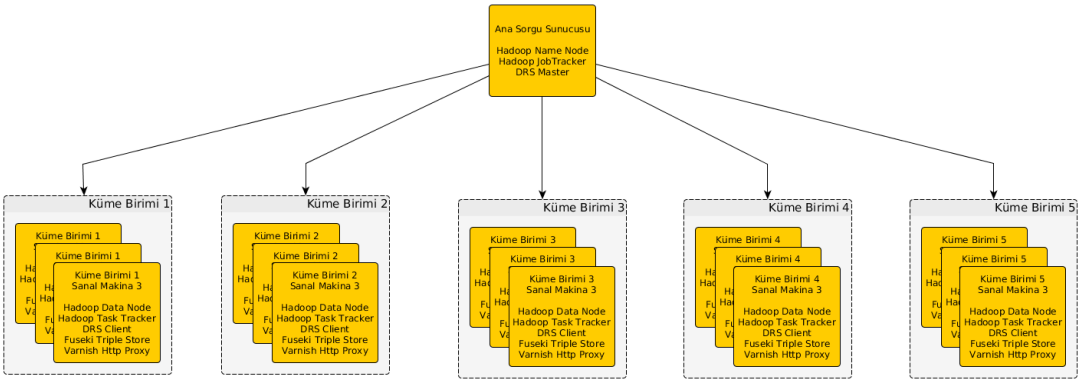

Şekil 3.2: Sistemin Sanal Makina Yapısı

Şekil 3.2'de görüldüğü üzere her bir fiziksel küme birimi üzerinde sanal makinalar çalışmaktadır. Bu sanal makinaların hepsi kendilerine ayrılmış bellek alanına ve işlemci kaynaklarına sahiptir. Deneyler sırasında, her bilgisayarda aynı sayıda olmak üzere, toplamda faklı sayıda sanal makinalar kullanılmıştır. Deneyler sırası ile, her bilgisayar üzerinde 1,2 ve 3 sanal makina çalışacak şekilde yapılmıştır. Her sanal makina bir işlemci çekirdeği kullanacak şekilde ayarlanmıştır. Bu sayede, fiziksel sistemdeki en düşük konfigürasyonlu bilgisayarın sahip olduğu toplam kaynakların aşılmaması ve sanal makinaların aynı anda çalıştırılmaları durumunda dahi fiziksel bilgisayar işletim sisteminin kullanabileceği işlemci ve bellek kaynağı kalması sağlanmaktadır. Bu tez çalışması kapsamında ölçülecek olan süreler, üretim ortamında kullanılacak gerçek sunucu bilgisayar konfigürasyonları ile elde edilecek olan sürelerden büyük ölçüde faklı olacaktır. Ancak çalışmada tespit edilmeye çalışılan en önemli noktanın oransal olarak nasıl bir kazanç elde edileceği olması dolayısı ile sanal makinalar deney amacı ile kullanılabilecektir. Fiziksel olarak ayrı gerçek sunucularda elde edilecek sonuçlar daha başarılı olacaktır.

Sistemin tasarımı sırasında mümkün olan her noktada açık kaynak olarak dağıtılan yazılımların kullanımının ve geliştirdiğimiz sisteme entegrasyonunun sağlanılmasına çaba sarf edilmiştir. Bu yaklaşımın sebebi, açık kaynak sistemlerin genel olarak çok iyi test edilmiş yazılımlar olmalarıdır. Ayrıca geliştirme sırasında ihtiyaç duyulan yardımın kullanıcı ve geliştiriciler tarafından çok hızlı sağlanıyor olması da başka bir sebeptir. Bu nedenlerle, sistemin uç birimlerinde veri depolamayı halihazırda çok başarılı şekilde yapabilen ve depoladıkları verinin yerel olarak çok hızlı bir şekilde sorgulanması için gerekli alt yapıyı sağlayan hazır yazılımlar kullanılmıştır. Bu sayede tez çalışması sırasında yapılan çalışmalar doğrudan dağıtık sistemin oluşturulmasına yönelik olup, uç birimlerin veriyi nasıl depolayacakları ve sorgulayacakları sorularını çözmeye gerek kalmamıştır. Bu yaklaşım genel olarak açık kaynak yazılımların amaçladığı kullanım biçimidir.

Sistemin çalıştıracağı yazılımlar açısından yapısı ayrı bir şekil olarak Şekil 3.3'de gösterilmiştir. Bu şekilde gösterilen tüm küme birimleri artık fiziksel değil sanal altyapıyı göstermektedir. Şekilde gösterildiği gibi, küme birimleri üzerinde bulunan her sanal makina üzerinde aynı konfigürasyon çalışmaktadır. Sistemler

arasındaki tek fark ağda kullandıkları IP adresleridir. Geliştirdiğimiz sistem için kullanılmakta olan bu küme yapısı aynı zamanda veri ön işleme safhasında kullanmakta olduğumuz Hadoop kümesini de içermektedir. Ayrıca tüm sanal makinalarda geliştirdiğimiz sisteme ait uç birim yazılımı bulunmaktadır. Verilerin depolanması ve sorgulanmasını sağlayan açık kaynak üçlü veritabanı ve SPARQL uç birimi yazılımı olan Jena Fuseki ve yapılan sorguların tekrar yapılması durumunda daha hızlı cevap verilmesini sağlayan önbellek yazılımı olan "Varnish HTTP Cache" yazılımı [1] çalışmaktadır.




Şekil 3.3: Sistemin Uygulama Dağılımı Yapısı

Sistemde kullanıcının doğrudan etkileştiği tek bilgisayar olan Ana Sorgu Bilgisayarı ise geliştirmiş olduğumuz yazılımın merkez birimini ve Hadoop master birimlerini içermektedir.

Sistemde kullanılan yazılımlar ve uygulama programlama arayüzleri hakkında kısa bilgiler ve sistemde yaptıkları görevler aşağıda verilmiştir.

### 3.1.1 DRS Merkez Uygulaması

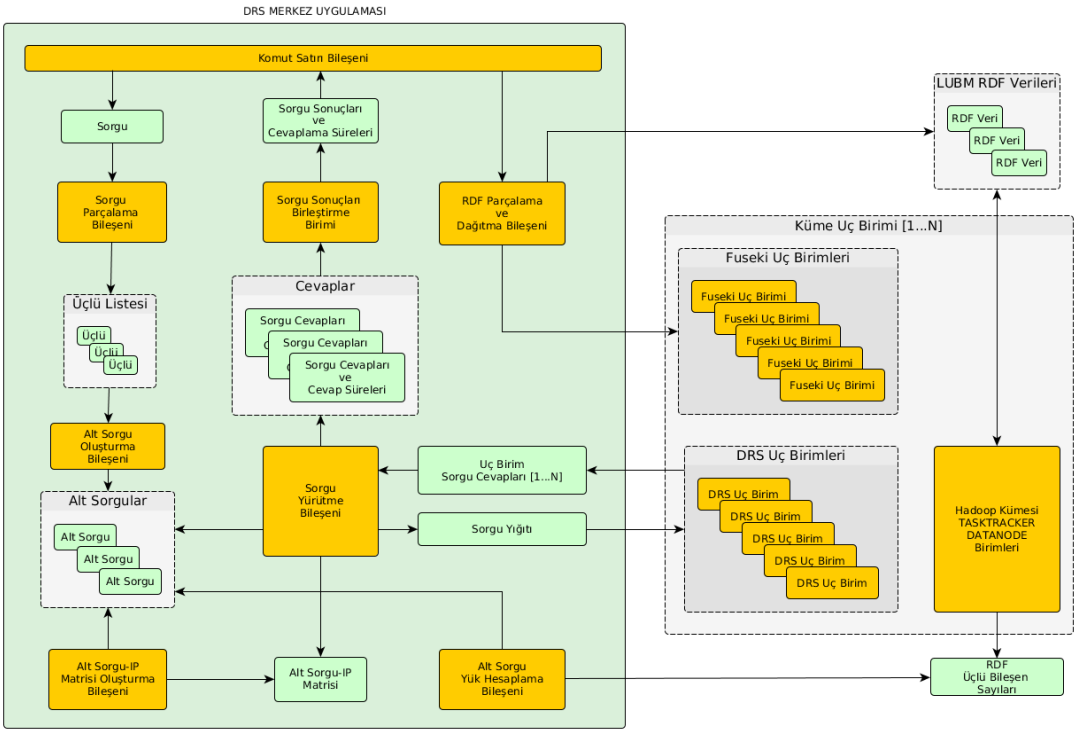
DRS (Distributed RDF Store), tez çalışması sırasında geliştirmiş olduğumuz uygulamanın kısaltılmış ismidir. DRS Merkez Uygulaması, Ana Sorgu Bilgisayarı üzerinde çalışmak üzere, Apache Jena Framework [2] kullanılarak Java

[1] https://www.varnish-cache.org/
[2] http://jena.apache.org

programlama dilinde yazılmıştır. Merkez Uygulaması, kullanıcının doğrudan etkileştiği ve yapılan sorguları alt sorgulara ayırarak, bunları küme uç birimleri üzerinde çalışan, DRS uç birim uygulamalarına gönderen ve elde edilen sonuçları kullanıcıya geri veren uygulamadır. Görevleri aşağıda liste olarak verilmiştir.

- Kullanıcı komut satır uygulamasını sunmak.
- Kullanıcının test amaçlı veri üretmesini sağlamak.
- Kullanıcının üretilen veriyi parçalamasını ve uç birimlere dağıtmasını sağlamak.
- Üretilen veri üzerinde, üçlü bileşen sayımı yapan Map-Reduce uygulamasını çalıştırmak.
- Kullanıcının SPARQL sorgu dosyalarını çalıştırmasını sağlamak.
- Sorguları parçalamak ve bu parçaları işletilmeleri için, parça ile ilgili uç birimlere göndermek.
- Sorgu cevaplarını kullanıcıya göstermek.
- Sorgu sürelerini ölçmek.

Şekil 3.4 DRS Merkez Uygulamasının alt bileşenlerini, bu bileşenler arasındaki veri akışını ve uygulamanın uç birimde bulunan diğer yazılım birimleri ile ilişkisini göstermektedir.




Şekil 3.4: DRS Merkez Uygulamasının Yapısı

DRS Merkez uygulaması farklı işleri yürüten alt programlardan oluşmuştur. Bunların en önemli olanları aşağıda kısaca açıklanmıştır.

#### 3.1.1.1 Komut Satırı Bileşeni

Komut satırı uygulaması, kullanıcının bilgisayarında bulunan terminal ekranından erişilebilen metin tabanlı bir arayüzdür. Kullanıcının yapabileceği işlemler için basit bir arayüz sağlamaktadır. Kullanıcının, LUBM veri üretici yazılımı aracılığı veri üretmesini, METIS uygulaması aracılığı ile üretilen veriyi parçalamasını ve bunların, küme uç birimlerinde bulunan Fuseki SPARQL sorgu uç birimlerine bağlı, Jena TDB veritabanlarına aktarılmasını ve SPARQL sorgu dosyalarını çalıştırmasını sağlayan komutları içermektedir.

#### 3.1.1.2 Sorgu Parçalama Bileşeni

Sorgu parçalama bileşenin amacı, sistem kullanıcısı tarafından verilen sorguların, alt sorgu oluşturma bileşeni tarafından kullanılacak olan parçalara ayrılmasını sağlamaktır. Sorgu parçalama bileşeni, kullanıcının verdiği sorgu dosyasını okur, okunan bilgilerin geçerli bir SPARQL sorgusuna ait olması durumunda, verilen sorguyu yapısal bileşenlerine ayrıştırarak, bir düz metin dosya çıktısı üretir. Sorgu parçalama bileşeni, arka planda, Rasqal Roqet [13] komut satırı uygulamasını kullanmaktadır. Rasqal Roqet uygulaması, verilen bir SPARQL sorgusunun bileşenlerine ayrılarak gösterilmesini sağlayan bir uygulamadır. Uygulamanın, komut satırından, DRS uygulaması aracılığı ile kullanılabilmesi için, Bash betikleri yazılmıştır. Bölüm-3 içinde, "SPARQL Sorgularının Parçalanması" başlığı altında, Rasqal Roqet komut satırı uygulamasının kullanılması ile, SPARQL sorgularının parçalanmasına bir örnek verilmiştir. Rasqal Roqet kullanımı ile elde edilen sorgu bileşenleri, sorgunun yüklemini (Query Verb), değişkenlerini (Query Bound Variables) ve üçlülerini (Triples) bir dosya halinde vermektedir. Sorgu parçalama bileşeni, bu değerlerin gerektiğinde kullanılabilmesini sağlamak amacı ile, dosya içindeki bileşenleri okur ve bunları uygulama içinde kullanılan değişkenlere atayarak hazırlanmasını sağlar.

#### 3.1.1.3 Alt Sorgu Oluşturma Bileşeni

Alt sorgu oluşturma bileşeninin amacı, kullanıcının komut satırı uygulaması aracılığı ile, girdi olarak verdiği sorgu dosyalarının çalıştırılması sırasında, küme uç birimlerine gönderilecek olan alt sorguları oluşturmaktır. Bu alt sorgular, kullanıcının verdiği sorgunun dağıtık ortamda yürütülmesi amacı ile üretilmektedirler. Bu bileşenler, alt SELECT sorgularının ve alt sorgu-IP matrisinin doldurulmasını sağlayan, ASK sorgularının oluşturulması sırasında kullanılmaktadırlar. Alt SELECT sorguları, kullanıcı tarafından verilen sorgunun üçlülerini sağlayan değerleri, küme uç birimlerinden almak için kullanılırken, ASK sorguları ise, uç birimlerin, kullanıcı sorgusuna ait üçlülerden hangilerini sağlayabildiğini bulmak için kullanılır. Bahsedilen SELECT ve ASK sorgularının

oluşturulması esnasında sorgu parçalama bileşeni tarafından sağlanan bileşenler kullanılır.

#### 3.1.1.4 Alt Sorgu-IP Matrisi Oluşturma Bileşeni

Alt sorgu oluşturma birimi tarafından oluşturulan ASK sorguları, kümeye ait uç birimlerden hangilerinin, kullanıcı tarafından girilen sorguya ait üçlüleri içerdiğini tespit etmek amacı ile kullanılırlar. Bu amaçla, kullanıcının girdiği sorguya ait değişkenler ve üçlüler kullanılarak hazırlanan ASK sorgusu, küme uç birimleri üzerinde çalışan Fuseki SPARQL uç birimlerine gönderilir. ASK sorgularına True veya False şeklinde cevaplar dönmektedir. Dönen değerler, indisleri sorgu üçlüsü ve IP olacak şekilde bir matris içine yazılır. ASK sorguları bütün küme uç birimlerine gönderilir ve DRS merkez uygulaması, ASK sorgularını, her biri ayrı bir izlek (Thread)olacak şekilde yürütür.

Bir Q sorgusundan türetilen ve 5 uç birime gönderilen, sq1, sq2, sq3 alt sorgularına (ASK) ait alt sorgu-IP matrisi çizelge 3.1 ile verilmiştir.

Çizelge 3.1: Genel bir Q sorgusuna ait örnek Alt Sorgu-IP matrisi

| **Alt Sorgu / IP** | **IP1** | **IP2** | **IP3** | **IP4** | **IP5** |
|---|---|---|---|---|---|
| **sq1** | 1 | 0 | 0 | 0 | 1 |
| **sq2** | 0 | 1 | 1 | 0 | 1 |
| **sq3** | 1 | 1 | 0 | 1 | 0 |

Çizelge 3.1içinde verilen örnek alt sorgu-IP matrisine göre, alt sorgu sq1, IP1 ve IP5 tarafından cevaplanabilirken, sq2 alt sorgusu IP2, IP3, IP5 ve sq3 alt sorgusu IP1, IP2 ve IP4 tarafından cevaplanabilmektedir. Tabloda satırlardan bir tanesinin sadece sıfır değeri içeriyor olması durumunda, o alt sorguya, küme tarafından cevap verilemediği anlamı çıkmaktadır. Bu durumda, sorgu yürütme birimi yığıt oluşturma işlemini durdurur ve kullanıcıya cevap bulunmadığı gösterilir.

#### 3.1.1.5 Alt Sorgu Yük Hesaplama Bileşeni

Bu yazılım birleşeni, alt sorgu oluşturma bileşeni tarafından oluşturulan, alt SELECT sorgularının, LUBM veri üretici yazılım tarafından üretilen RDF dosyaları kullanılarak cevaplanması durumunda, verilecek cevapların büyüklüğü için olası en yüksek değeri hesaplar. Hesaplanan bu değerin, veri parçalanarak depolandığı için, cevapların pratikte tek seferde ve tek küme uç biriminden elde edilememe ihtimalini de göz önünde bulundurarak, bir yaklaşım olduğu değerlendirilmelidir. Ancak, verilen cevaplar en iyi ihtimal ile, sadece tek küme uç biriminden verilebilirken, en kötü ihtimal ile tüm uç birimler cevabın bir kısmını sağlayacaktır. Cevaplamanın, mümkün olduğunca tek küme uç birimi tarafından yapılmasını sağlamak amacı ile, METIS çizge parçalama yazılımı kullanılmıştır. Alt SELECT sorgularına verilecek cevapların, en iyi ihtimal ile tek küme uç biriminden verilmesi durumunda, cevabın büyüklüğü oluşacak olan ağ trafik yoğunluğu ile orantılı olacaktır. Bu durumda büyük cevapların, küme uç birimleri arasında gerçekleşecek olan haberleşmelerde, en az sayıda iletilmesi sistem performansına pozitif bir katkı sağlayacaktır. Bu açıdan, alt sorgu gerçekleşmeden, dönmesi olası olan cevap büyüklüklerinin teorik en yüksek değerlerinin bilinmesi, sorguların belli bir sır içerisinde çalıştırılmasını sağlayacaktır. Alt sorgu yük hesaplama birimi bu işlemi, bir Hadoop Map-Reduce uygulaması kullanılarak oluşturulmuş, konu-yüklem-nesne (subject-predicate-object) frekans listesini kullanarak belirlemektedir [3]. Çıktı olarak alt SELECT sorgularının yük değerlerinin artan değerlerine göre sıralanmış bir liste oluşturmaktadır. Alt sorgu yük hesaplama sürecinin detayları Bölüm-3 altında bulunan "Alt SPARQL Sorgularının Yük Değerlerinin Hesaplanması" başlığı altında anlatılmıştır. Örnek olarak, verilen genel bir Q SPARQL sorgusuna ait alt SELECT sorguları olan sq1, sq2, sq3 için, [sq1,sq3,sq2] şeklinde bir liste dönmektedir. Bu sonuç, sq1 alt sorgusunun yük değerinin en az, sq2 alt sorgusunun yük değerinin en yüksek olduğunu göstermektedir. Bu liste, sorgu yürütme birimi tarafından hazırlanan sorgu-IP yığıtının oluşturulması sırasında kullanılmaktadır.

[3] Hadoop tarafından oluşturulan bu çıktı dosyasına ait bir bölüm, Ek-D içinde bulunan "RDF üçlü bileşen sayıları" başlığı altında örnek olarak verilmiştir.

#### 3.1.1.6 Sorgu Yürütme Bileşeni

Sorgu yürütme bileşeni, alt sorgu-IP matrisi oluşturma birimi tarafından kullanılmış olan ASK sorgularına karşılık gelen ve alt sorgu oluşturma birimi tarafından hazırlanan SELECT sorgularını, küme uç birimlerinde bulunan DRS uç birim uygulamalarına gönderen yazılım birimidir. Alt sorgu oluşturma birimi, ASK sorgularını oluşturmak amacı ile kullandığı değişkenleri ve üçlüleri, SELECT sorgularını hazırlamak amacı ile de kullanır. DRS uç birim uygulamalarına gönderilecek olan bu sorgular, bir yığıt (Stack) veri yapısı içinde, hangi IP'lerde çalıştırılacağı bilgisi ile birlikte tutulmaktadır. Alt sorguların hangi IP'lerde çalıştırılacağı bilgisi, alt sorgu-IP matrisi oluşturma birimi tarafından üretilen matris kullanılarak belirlenir. Sorgular, bu yığıta, alt sorgu yük hesaplama bileşeni tarafından hesaplanan, yük değerlerinin artma sırasına göre eklenir. Örneğin, genel bir Q SPARQL sorgusuna ait sq1, sq2 ve sq3 alt sorgularının, alt sorgu yük hesaplama bileşeni tarafından değerlendirilmesi sonucunda oluşturulan çıktının [sq1,sq3,sq2] şeklinde olması ve bu genel Q sorgusu için alt sorgu-IP matrisinin çizelge 3.1'de verildiği gibi hesaplanması durumunda, oluşturulacak olan yığıt [sq2-IP2, sq2-IP3, sq2-IP5, sq3-IP1, sq3-IP2, sq3-IP4, sq1-IP1, sq1-IP5] şeklinde olacaktır. Bu yığıt, sorguların hangi sıra ile, hangi uç birimler üzerinde çalıştırılacağının bilgisini içermektedir. Alt SELECT sorguları sq1, sq2 ve sq3 olan genel Q sorgusu için, bu yığıt içinde bulunmayan uç birimler kullanılmayacaktır. Yığıtın içindeki alt sorguların işletilmesi sürecinde, yığıt uç birimler arasında yapılacak olan haberleşme sırasında gönderilip alınacak ve ilgili her uç birimde değişikliğe uğradıktan sonra, sıradaki diğer uç birime gönderilecektir. Yapılan değişiklik, yığıt içinde en üstte bulunan ve uç birimin kendisi için hangi sorgunun cevaplanacağını gösteren elemanın, yığıttan çıkarılmasıdır. Örneğin, burada örnek olarak gösterilen yığıtın, sorgu yürütme bileşeni (DRS Merkez Uygulaması bileşeni) tarafından gönderilmesi sırasında uğradığı değişiklikler sıra halinde aşağıdaki tabloda verilmiştir.

Çizelge 3.2: Genel bir Q sorgusuna ait alt sorgular ile oluşturulan sorgu yığıtının, farklı küme uç birimlerinde işlenişi

| No | Alt Sorgu | Uç Birimler | Yığıt İçeriği |
|---|---|---|---|
| **1** | - | DRS Merkez Uyg. | [sq2-IP2, sq2-IP3, sq2-IP5, sq3-IP1, sq3-IP2, sq3-IP4, sq1-IP1,sq1-IP5] |
| **2** | sq2 | IP2 | [sq3-IP1, sq3-IP2, sq3-IP4, sq1-IP1, sq1-IP5] |
| **3** | sq2 | IP3 | [sq3-IP1, sq3-IP2, sq3-IP4, sq1-IP1, sq1-IP5] |
| **4** | sq2 | IP5 | [sq3-IP1, sq3-IP2, sq3-IP4, sq1-IP1, sq1-IP5] |
| **5** | sq3 | IP1 | [sq1-IP1, sq1-IP5] |
| **6** | sq3 | IP2 | [sq1-IP1, sq1-IP5] |
| **7** | sq3 | IP4 | [sq1-IP1, sq1-IP5] |
| **8** | sq1 | IP1 | [] |
| **9** | sq1 | IP5 | [] |

Bu çizelgede dikkat edilecek olan durum, aynı işlemin birden fazla uç birimde aynı anda gerçekleşiyor olduğudur. Örneğin, sq3 alt sorgusunu cevaplayabilen IP2, IP3 ve IP5 uç birimlerine aynı yığıt gönderilmiş ve hepsi, paralel olarak sq3 ile ilgili kısımları kendileri işleyecekleri için, yığıttan ayrı ayrı çıkartmışlardır.

Yığıt oluşturulduktan sonra, sorgu yürütme bileşeni, yük değerlerine göre dizilmiş alt sorgu listesindeki ilk sorguyu (bu örnekte sq2) cevaplayabilecek olan IP adreslerine (bu örnek için IP2, IP3 ve IP5) yığıtı gönderir. Burada önemli olan nokta, her küme uç biriminin, yığıt içinde sadece kendisi için olan kısımları işleyecek ve değiştirecek olmasıdır. Kendi işleyebileceği kısım dışındaki yığıt elemanları için, ilgili uç birime yığıtın son halini gönderecektir. Sorgu yürütme bileşeni, gönderme işlemi ile birlikte süre hesaplaması için, cevapların gelmesine kadar geçecek süre boyunca çalışacak olan sayacı başlatır. Bu noktada, merkez DRS uygulaması cevapların gelmesine kadar ki geçen sürede başka faaliyet yürütmez. Cevapların gelmesi ile birlikte, sayaç değeri okunur ve bu değerler, gelen cevaplarla birlikte, sorgu sonuçları birleştirme birimine gönderilir. Kaç uç birimden cevap geleceği, oluşturulan yığıtın, en fazla yük değerine sahip olan sorgusunu cevaplayabilen IP sayısına bağlıdır. Bu değer en düşük sıfır olabilirken, en fazla küme büyüklüğü kadar olacaktır. Yığıtın küme içinde işlenmesi sırasında izlenen adımların detayları Bölüm-3 içinde yer alan, "Sorguların Dağıtık Ortamda Yürütülme Süreçleri" başlığı altında anlatılmıştır.

#### 3.1.1.7 Sorgu Sonuçları Birleştirme Birimi

Sorgu sonuçları birleştirme birimi, sorgu yürütme birimi tarafından görevlendirilen küme uç birimlerinden gelen cevapları ve bu cevapların, sorgu yürütme birimine geri dönmesi için gereken zaman değerlerinin birleştirilmesi işlemini yürütür. Birleştirilen sonuçlar komut satırı bileşenine gönderilerek kullanıcıya gösterilmesi sağlanır. Sorgu sonuçlarının birleştirilmesi amacı ile izlenen adımlar, Bölüm-3 içinde yer alan, "Alt SPARQL Sorgu Sonuçlarının Birleştirilmesi" başlığı altında detaylı olarak anlatılmıştır. Hesaplanan zaman değerleri, sorgu yığıtının, sorgu yürütme bileşeninden ilgili uç birimlere gönderilmesi ve bunlara gelen cevapların alınması arasında geçen zamandır.

### 3.1.2 DRS Uç Birim Uygulaması

DRS Uç Birim uygulaması, DRS uygulamasının, küme uç birimleri üzerinde çalışan kısmıdır. Apache Jena Framework kullanılarak, Java programlama dilinde uygulaması yapılmıştır. Genel olarak görevi, DRS Merkez Uygulaması tarafından, kurulu olduğu küme birimine gönderilen alt sorgu yığıtı içindeki ilgili alt sorguları çalıştırmak, ara sonuçların birleştirilmesini sağlamak ve sonuçları merkez uygulamaya geri göndermektir. Gerçekleştirdiği görevler aşağıda listelenmiştir.

- DRS Merkez uygulamasından gelen, alt sorgu yığıtı içinde bulunan ve yerel olarak çalıştırılması gereken sorguları yürütmek.
- Yerel olarak yürütülen alt sorguya ait, yığıt içinde bulunan elemanları silmek.
- Yığıt içinde bulunan sıradaki sorguların cevaplanabilmesi için, yığıtı, sırada bulunan sorguyu cevaplayacak ilgili uç birimlerine göndermek.
- Diğer uç birimlere gönderilen alt sorgu yığıtı içinde bulunan ve gönderildikleri uç birimler tarafından çalıştırılacak olan sorguların cevaplarını almak ve yerel olarak çalıştırılan sorgu cevapları ile birleştirmek.

- Elde edilen ara sonucu, kendisine yığıtı gönderen uç birime veya DRS Merkez uygulamasına göndermek [4].

DRS uç birim uygulamasının, görevleri açısından özelleşmiş bileşenlerinin kısa açıklaması aşağıda verilmiştir.

#### 3.1.2.1 Sorgu Yönetme Bileşeni

Sorgu yönetme bileşeni, DRS merkez uygulaması veya başka bir küme uç birimi tarafından gönderilen, alt sorgu yığıtını alarak işleyen ve bunlara karşılık olarak elde ettiği cevapları gönderen yazılım birimidir. Alınan yığıt içinde en üst seviyede bulunan alt sorgunun, yerel olarak çalıştırılması için, yığıt içinden çıkarılmasını sağlar. Eğer, en üst seviyede bulunan alt sorgu, yığıta birden fazla uç birim IP'si ile eklenmişse, sadece üzerinde bulunduğu IP değeri için olan yığıt elemanı değil, diğer IP'ler için eklenmiş olan elemanlarda yığıttan çıkarılır. Bu şekilde aynı alt sorguyu içeren yığıt elemanlarının hepsi yığıttan çıkarılmış olmaktadır. Çıkarılan alt sorgu, yerel olarak yürütülmesi amacı ile, aynı küme uç birimi üzerinde çalışmakta olan Fuseki SPARQL uç birimine gönderilir. Sorguya verilen cevap, sorgu sonuçları birleştirme bileşeni tarafından, diğer uç birimlerden gelen cevaplar ile birleştirilmek üzere alınır. Cevaplar birleştirildikten sonra geri gönderilme işlemini Sorgu yönetme birimi yapmaktadır. Yığıtın değiştirilmiş hali, sırada bulunan sorguyu cevaplayabilecek olan diğer küme uç birimlerine, sorgu delege bileşeni aracılığı ile gönderilir. Sıradaki sorguyu cevaplayabilecek olan uç birimlerin IP'leri yığıt içinden okunmaktadır.

[4] Cevapların hangi uç birime geri gönderileceği, cevabı hazırlayan uç birimin, yığıt içindeki sorgu cevaplama hiyerarşisinde bulunduğu konuma göre değişecektir. Cevabı gönderecek olan uç birim, eğer, DRS Merkez uygulamasının yığıtı gönderdiği ilk uç birimlerden biri ise, DRS Merkez Uygulamasına, aksi takdirde, kendisine değiştirilmiş yığıt gönderen diğer uç birime cevap gönderecektir.

#### 3.1.2.2 Sorgu Delege Bileşeni

DRS uç birim uygulamasına, DRS merkez uygulamasından veya diğer küme uç birimleri tarafından gönderilen alt sorgu yığıtı, sorgu yönetme bileşeni tarafından işlendikten sonra, değiştirilmiş olan yığıtın, sıradaki sorguyu cevaplayacak uç birimlere gönderilmesini sağlayan bileşen, sorgu delege bileşenidir. Sorgu delege bileşeni, değiştirilmiş yığıtın gönderileceği uç birimlerin listesini, yığıtta, sırada bulunan alt sorguyla eşleştirilmiş IP'leri tespit ederek elde etmektedir. DRS merkez uygulaması üzerinde çalışan, alt sorgu-IP matrisi oluşturma bileşeni tarafından oluşturulan matrise bağımlı olarak, gönderim yapılacak IP adresi sayısı değişebilir. Bu değer 1 ile küme büyüklüğü arasında değişmektedir. Değerin sıfır olması durumu, DRS merkez uygulamasında, yığıt oluşturulması sırasında ele alınmakta ve değer sıfır ise yığıt oluşturulmamaktadır. Çünkü bu durum, alt sorgulardan bir tanesine, hiç bir küme uç biriminin cevap veremediği anlamına gelmektedir ki, bu durumda sorgunun geri kalan kısımlarının yürütülmesine gerek kalmayacaktır.

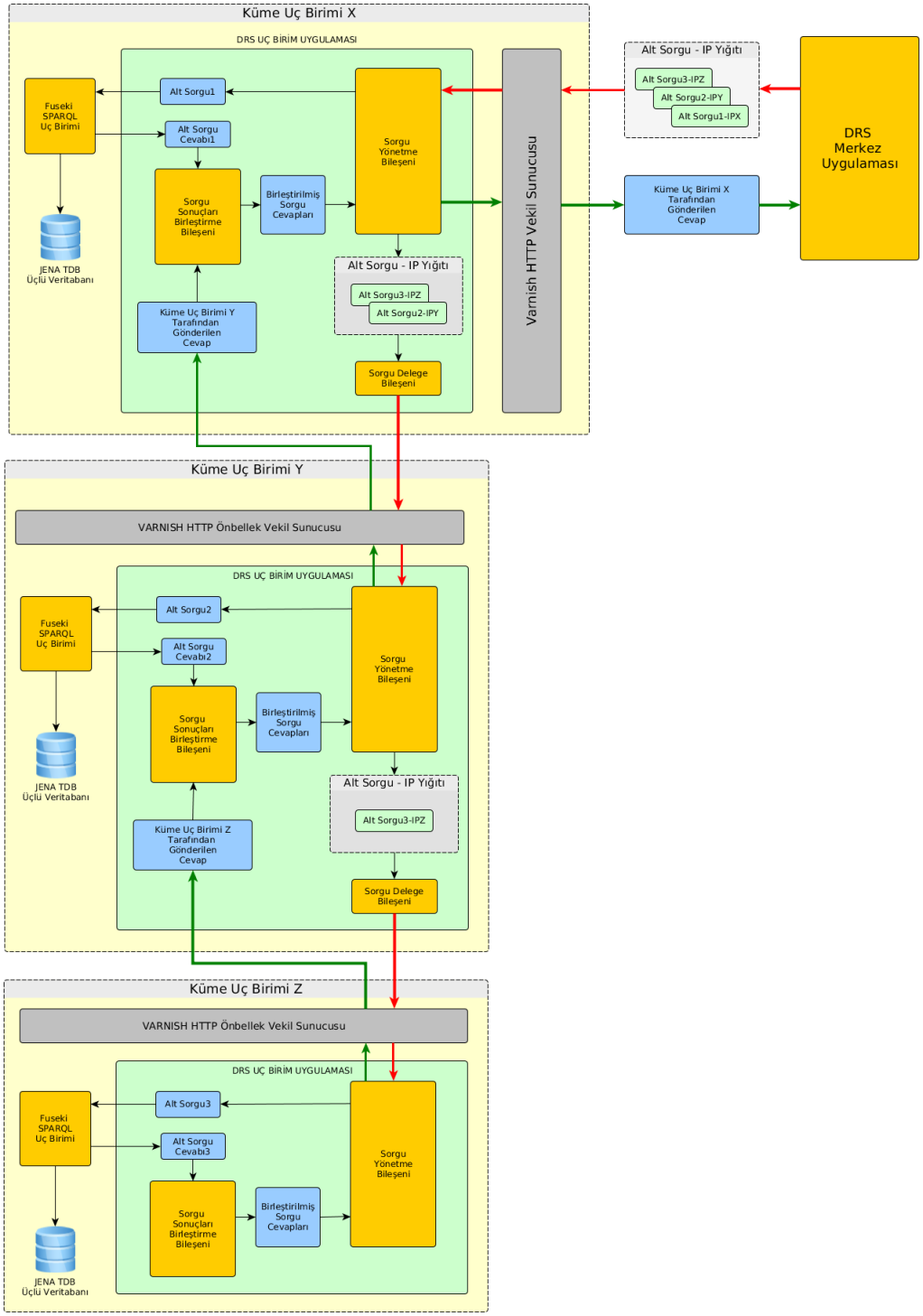
#### 3.1.2.3 Sorgu Sonuçları Birleştirme Bileşeni

Yerel olarak, Fuseki SPARQL uç birimine gönderilerek çalıştırılan sorguların cevaplarını ve diğer uç birimlere delege edilmiş sorgu yığıtları için elde edilen sonuçları birleştirme görevini yürüten yazılım bileşenidir. Görevi açısından, DRS merkez uygulaması içinde bulunan sorgu sonuçları birleştirme bileşeni ile aynı görevleri yürütmektedir. Birleştirilmiş olan sonuçları, sorgu yığıtını kurulu olduğu küme uç birimine gönderen birime geri gönderilmesi amacı ile, sorgu yönetme bileşenine iletir. Sorgu sonuçlarının birleştirilmesi hakkında detaylı bilgi Bölüm-3 içinde yer alan, "Alt SPARQL Sorgu Sonuçlarının Birleştirilmesi" başlığı altında verilmiştir[5].

DRS uç birim uygulamasının yapısı ve diğer DRS uç birimleri ile haberleşme

[5]Sorgu sonuçlarının, küme uç birimlerde ara birleştirme işlemlerine sokulmaları, birleştirme işleminin küme kesişimlerine dayanan yapısı dolayısı ile, ağ üzerinde aktarılan verinin mümkün olduğunca azaltılması açısından önemli bir rolü vardır.

biçimi Şekil 3.5'de verilmiştir.

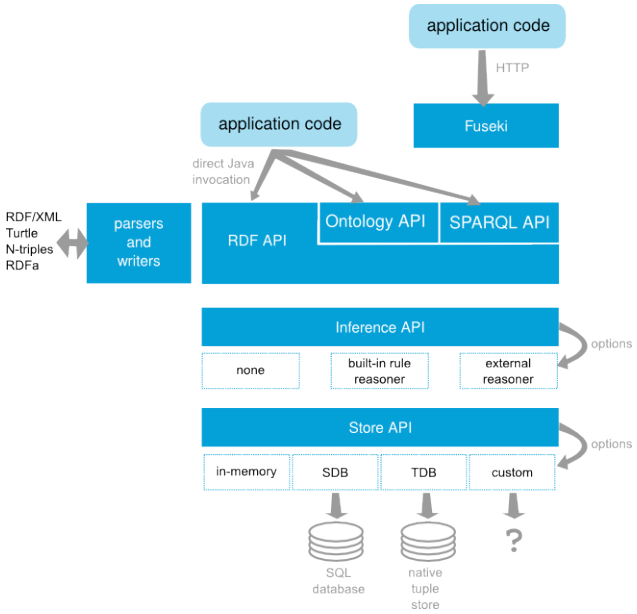



Şekil 3.5: DRS uç birim uygulamasının yapısı ve uç birimlerin haberleşmesi

### 3.1.3 Apache Jena Uygulama Geliştirme Çatısı (Framework)

Apache Jena Framework, Java programlama dili ile geliştirilmiş, açık kaynak kodlu bir semantik ağ uygulama geliştirme çatısı yazılımıdır. Bu uygulama geliştirme çatısı, RDF API, SPARQL API ve OWL API bileşenlerinden oluşmakta ve semantik uygulamaları geliştirmek için çok detaylı bir kütüphane sunmaktadır. Apache lisansı ile dağıtılıyor olması dolayısı ile her türlü ticari uygulamada kullanılması mümkündür. Çok detaylı bir dokümantasyona sahip olması, Jena göz önüne alınarak yazılmış Semantik Ağ kitaplarının bulunması ve çok aktif bir geliştirici kitlesine sahip olması dolayısı ile, tez çalışması sırasında, bu kütüphanenin kullanılmasına karar verilmiştir [10, 25].

Jena uygulama geliştirme çatısına ait mimari yapı Şekil 3.6'da verilmiştir.




Şekil 3.6: Jena Uygulama Geliştirme Çatısına ait mimari yapı [6]

[6] http://jena.apache.org/getting_started/index.html

### 3.1.4 Apache Jena Fuseki SPARQL Uç Birimi

Fuseki, Jena Projesi kapsamında dağıtılan bir SPARQL uç birimidir [7]. Tez çalışmasında tasarladığımız sistemde uç birimler üzerinde çalışarak, gelen SPARQL sorgularının yerel Jena TDB üçlü veritabanından yanıtlanmasını sağlayacaktır. Fuseki'nin çalışmalarda kullanılmak üzere seçilmesinini en önemli nedeni, Fuseki'nin HTTP ile gönderilen sorguları cevaplayabilme ve cevapları isteğe bağlı olarak JSON, XML veya Text formatında gönderme yeteneği olmasıdır. Ayrıca, istenildiği takdirde, sağladığı web arayüzü ile doğrudan kullanıcılar tarafından da sorgulanabilmektedir. Bu web arayüzü, veri yükleme amacı ile de kullanılabilmektedir. Kurulduğu bilgisayar üzeride, komut satırından yönetiminin sağlanması amacı ile çeşitli komutlar sağlamaktadır.

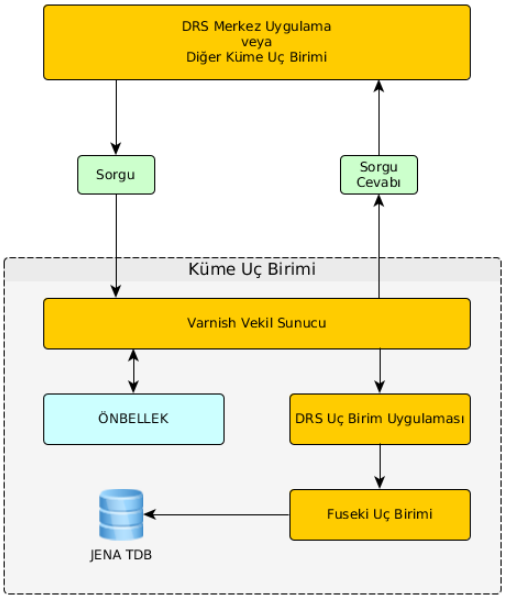
### 3.1.5 Varnish Http Önbellek Vekil Sunucusu

Sistemdeki tüm uç birimlerde Varnish Http Önbellek Vekil sunucusu [8] çalışmaktadır. Tek bir kullanıcı sorgusu sonucunda, oluşturulan alt sorgular herhangi bir uç birim üzerinde birden fazla defa çalıştırılabilmektedirler. Bu durumda, uç birimde bulunan Fuseki SPARQL cevaplama biriminin, aynı alt sorguyu veritabanından defalarca okuması gerekecektir. Bu durumu engellemek için her uç birime, gelen sorguları ve bu sorgulara verilen cevapları önbellekte tutan bir vekil sunucusu kurulması tasarlanmıştır. Bu sayede, her uç birim kendisine gelen sorgu ve cevaplarını bellekte ayrılan alandan okuyarak göndermekte ve bir veritabanı işlemi yapılması engellenmektedir. Tez çalışması için, birçok vekil sunucu içinden Varnish Vekil Sunucunun seçilmesinin en önemli nedeni, Varnish'in önbellek olarak işletim sistem fiziksel belleğini kullanıyor olması ve kullanılan bu alanın büyüklüğünün kullanıcı tarafından belirlenebiliyor olmasıdır. Alanın büyüklüğünün ayarlanabiliyor olması sayesinde, önbellek olarak kullanılacak olan alanın, sadece tek kullanıcı sorgusu sonucunda oluşabilecek alt sorgulara yetecek seviyede ayarlanması, dolayı ile de önbellek aramalarının sadece tek

[7] http://jena.apache.org/documentation/serving_data/
[8] https://www.varnish-cache.org/

kullanıcı sorgusu için yapılması sağlanabilmektedir. Bu da önbellek arama süreleri açısından zaman tasarrufu sağlamaktadır.

Varnish'in tasarlanan sistemdeki yeri Şekil 3.7'da gösterilmiştir.




Şekil 3.7: Varnish Http Önbellek Vekil Sunucusu

### 3.1.6 Apache Hadoop

Tez çalışması içinde alt sorguların dönecekleri cevap miktarları hakkında bir ön değerlendirme yapabilmesi ve buna dayanarak da sorguların sıralamasının yapılabilmesi için, RDF dosyalarında geçen her özne, yüklem ve nesne bileşenlerinin sayılması gerekmektedir. Bu işlem Hadoop üzerinde gerçekleştirilmektedir. Hadoop kurulumu ile ilgili detaylı bilgiler Ekler bölümünde Ek-E içinde verilmiştir.

Hadoop geniş ölçekli bir dağıtık işleme altyapısıdır [47, 54, 33]. Tek bir bilgisayar üzerinde çalıştırılabileceği gibi, yüzlerce hatta binlerce düğüm üzerinde de çalıştırılabilmektedir. Asıl gücü bir küme düğüm üzerinde çalıştırıldığında görülebilmektedir. Çok büyük işlem yükü olan problemler düğümlere dağıtılarak

işlenmiş sonuçların birleştirilmesi esasına göre çalışmaktadır. Burada bahsedilen veri büyüklüğü ile genelde petabyte-terabyte seviyesindeki veriler kastedilmektedir. Hadoop bu ölçekteki verileri işlemek üzere tasarlanmış ve geliştirilmiştir. Bu ölçekte bir verinin tek bir bilgisayar üzerinde işlenmesi çok uzun zaman almaktadır.

Farklı dağıtık işlem çözümleri problemlerin çözümü sırasında farklı noktaları göz önünde tutarlar. Bazıları yüksek güvenliğin en önemli parametre olduğu yönünde yaklaşımlar ortaya koyarken bazıları hatalara karşı daha dayanıklı çözümler ortaya koymaktadırlar. Bu parametreler çok çeşitli olabilir, ancak Hadoop tasarımı verinin paralel olarak hızlı bir şekilde işlenmesi, donanım arızalarının sebep olduğu aksaklıkların daha iyi yönetilmesi ve mevcut donanım ve ağ kaynakların kullanım oranlarının daha başarılı bir şekilde iyileştirilmesi amaçları göz önüne alınarak tasarlanmıştır.

#### 3.1.6.1 Hadoop Veri Dağıtımı

HDFS, verinin yüklenme aşamasında, küme düğümlerine dağıtılmasını sağlamaktadır. Küme içindeki tüm düğümler, veri saklama işlemi için kullanılırlar. Hadoop, büyük veri kümelerini küçük parçalara ayırarak, her bir düğümün verinin belli bir bölümünü depolamasını sağlamaktadır. Düğümlerden herhangi birinde arıza oluşması durumunda veri kaybının oluşmaması için her bir veri parçası birden fazla düğüm üzerine kopyalanır. Bu açıdan Hadoop veri depolama açısından hata toleransı çok yüksek bir yapıdır. Hangi veri parçasının hangi düğüm üzerinde bulunması gerektiğine Hadoop otomatik olarak karar vererek, bu bilgiyi veri işleme görevlerinin kullanabilmesi için depolamaktadır. Hadoop verinin işlem yapılacak olan düğüme taşınması yerine, işlemlerin veriye en yakın düğüm üzerinde çalışması ilkesine göre tasarlanmıştır. Bu özellik çok büyük veri kümelerinin işlenmesi sırasında verinin düğümler arasında ağ trafiği oluşturmasını engeller.

#### 3.1.6.2 Hadoop Ölçeklenebilirliği

Hadoop'un diğer paralel işleme sistemlerine göre en büyük avantajlarından biriside kolay ölçeklenebilirliğidir. Bir Hadoop kümesinin sağlayacağı işleme gücü, kümedeki düğüm bilgisayar sayısına bağlıdır. Kümedeki bilgisayar sayısı az iken Hadoop uygulamalarının çalıştırılması için geçen zaman masrafları dikkate alınacak olunursa, diğer paralel işleme sistemlerinde daha düşük bir performans gözlenecektir. Ancak düğüm bilgisayar sayısının arttırılması durumunda, Hadoop'un diğer sistemlere oranla işlemleri tamamlama süresinin çok daha az olduğu gözlemlenebilecektir.

#### 3.1.6.3 Hadoop MapReduce Programlama Yaklaşımı

Hadoop kümesi üzerinde çalıştırılacak olan programlar MapReduce programlama modeline göre yazılmış olmalıdır [9]. Hadoop, herhangi başka bir biçimde yazılmış diğer programları küme üzerinde paralel olarak çalıştıramaz. Veri parçalarını, bir düğüm üzerinde diğerlerinden yalıtılmış olarak işleyerek ara çıktıları oluşturan işlemlere Mapper denir. Mapper işlemleri, her seferinde, girdi olarak verilen dosyaların bir parçasını işlerler. Bir Hadoop uygulamasında yaptığı iş açısından farklı Mapper işlemleri olabilir. Mapper işlemleri tarafından oluşturulan ara çıktılar birer anahtar-değer ikilisi şeklindedir. Hadoop kümesi üzerinde hangi birimlerde, ne zaman ve hangi Mapper işleminin çalışacağına, Hadoop JobTracker birimi karar vermektedir. Farklı küme birimlerinde oluşturulan bu ikililer, Shuffle-Sort şeklinde isimlendirilen bir ara süreç içinde, ikilinin anahtarına göre gruplanır ve sıraya dizilirler. Oluşturulan bu ara çıktılar Reducer isimli başka işlemlere girdi olarak gönderilirler. Mapper işlemlerine benzer şekilde, hangi küme biriminde, ne zaman, hangi Reducer işleminin çalıştırılacağına JobTracker karar verir. Hadoop, küme düğümleri arasındaki tüm veri akışını ve kümenin topolojik yapısını kendisi yönetmektedir. Bu açıdan, MPI ve benzeri, daha eski paralel programlama yöntemlerindeki gibi, veri akışının soketler üzerinden, doğrudan programcı tarafından yazılan programlar ile yönetilmesine gerek yoktur. Düğümlerden

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

herhangi birinin arıza yapması durumunda, o düğüm üzerinde çalışması gereken işlemler başka bir düğüm üzerinde çalıştırılmak üzere yeniden planlanırlar. Bu işlemler tamamen Hadoop JobTracker birimi tarafından yönetilirler. Her bir düğüm diğerlerinden izole bir şekilde çalıştıkları için, başka bir düğüm üzerinde oluşan arızadan etkilenmeden görevlerini yapmaya devam ederler.

#### 3.1.6.4 HDFS

HDFS, Hadoop tarafından kullanılmakta olan dağıtık dosya sisteminin ismidir. HDFS kısaltması, "Hadoop Distributed File System" (Hadoop Dağıtık Dosya Sistemi) ifadesindeki kelimelerin baş harflerinden oluşmaktadır. Tüm dağıtık dosya sistemlerinde olduğu gibi, HDFS için de en önemli tasarım amaçları, terabyte-petabyte ölçeğinde verilerin depolanabilir olması ve bu veriye ağ üzerindeki farklı noktalardan erişilebilir olmasıdır.

HDFS dışında başka dağıtık dosya sistemleri de bulunmaktadır. Ancak her biri amaçlanan kullanım alanına özgü avantajlara ve dezavantajlara sahip olacak şekilde tasarlanmışlardır.

HDFS tasarımının getirdiği bazı kısıtlar bulunmaktadır.

1. HDFS kullanan uygulamaların, dosyalardan uzun sıralı akan okumalar (long sequential streaming read) yapacakları düşünülmüş ve HDFS bu tür okumalar için geliştirilmiştir. Bu yüzden rastgele arama bazlı okumalarda zaman açısından düşük performans göstermektedir.

2. Verilerin HDFS'e bir kere yazılması ve sonra tekrarlayan sayılarda okunacağı öngörülmüştür. Bu açıdan veri silme ve güncelleme HDFS üzerinde desteklenmemektedir.

3. Verilerin büyüklükleri ve akan şekilde okunmaları dolayısı ile yerel olarak saklanması (local caching) desteklenmemektedir. Yerel bir kopya oluşturulması için harcanacak olan kaynak ve çaba, verinin HDFS kaynaklarından tekrar okunmasından daha fazla olmaktadır.

4. Arıza yapan küme düğümleri üzerindeki işlerin diğer küme düğümlerine dağıtılacak olması dolayısı ile, performansta kayıp olacak olmasına rağmen, sistemin görevini yerine getirmesi sağlayacak şekilde kaynak yönetimi yapılmaktadır.

HDFS blok yapılı bir dosya sistemidir. Dolayısı ile, boyutları blok büyüklüğünden daha fazla olan tüm dosyalar, dosya sisteminin blok büyüklüğüne sahip küçük parçalara bölünürler ve bu parçalar küme düğümleri üzerine dağıtılarak saklanırlar. Bir dosya birçok bloktan oluşabileceği ve bloklar farklı makinalarda depolandığı için, sadece bir dosyanın sisteme sunulması birden fazla düğümüm birlikte çalışması ile mümkün olmaktadır. Blokları depolayan düğümlerde oluşabilecek olan arızlar dolayısı ile dosyanın bir kısmının okunamaz hale gelmesi, tüm dosyanın kaybedilmesi anlamına gelir. Bu problemi ortadan kaldırmak için, HDFS, her bir bloğun birden fazla düğüm üzerine kopyalanmasını sağlar. Burada kullanılan, varsayılan küme sayısı, 3 olarak ön tanımlı şekilde tasarlanmıştır ve değiştirilebilir. Ancak kopyalanacak düğüm sayısının arttırılması hem ağ kaynaklarının daha çok kullanılmasına, hem de sistemin toplamda sahip olduğu depolama kaynaklarının verimsiz bir şekilde harcanmasına sebep olur.

NTFS veya ext3 gibi sıradan dosya sistemlerinde blok boyutları 4 veya 8 KB iken, bu değer büyük dosyaları işlemek amacı ile tasarlanan HDFS üzerinde 64Mb'dır. Bu sayede, blokların düğümler üzerinde dağılımları ile ilgili bilgileri depolamak amacı ile kullanılan üst-verilerin (metadata) boyutlarının küçük olması sağlanır.

Veri depolamak amacı ile kullanılan düğümlerde (Linux işletim sistemi) sistemin dosya sistemi ile düğümler arasında ortak olarak kullanılan dağıtık HDFS dosya sistemi birbirinden farklıdır. Düğüm bilgisayarlardan bir tanesine giriş yapılıp terminal ekranında bir komut verilmesi durumunda gösterilecek olan sonuç, normal dosya sistemi üzerinde çalıştırılarak döner. HDFS başka bir isim uzayında olduğu için, verilen komutların, varsayılan isim uzayındaki sonuçları gösterilir. Dolayısı ile HDFS üzerinde depolanan veriler üzerinde normal Linux terminal komutları ile bir işlem yapılamaz. Ancak HDFS, sıradan komut kümesinin tamamen paralel olarak geliştirilmiş başka bir komut kümesi ile dağıtılmaktadır.

Bu komutlar ile normal dosya sisteminde olduğu gibi HDFS dosya sistemi üzerindeki dosyalar ile işlemler yapılabilmektedir.

## 3.2 Sistemin Çalışması

Bu bölümde, mimari yapısı hakkında bilgi verilmiş olan tasarımın çalışması detaylı olarak açıklanacaktır. İlk olarak, tasarımı yapılan bu sistemin veri üzerinde yaptığı ön işlemler anlatılacaktır. Sonraki bölümde verinin parçalanması ve sorgulanmak üzere depolanacakları uç birimlere dağıtılması işlemleri anlatılacaktır. Son olarak, kullanıcı tarafından çalıştırılan bir sorgunun, dağıtık hale getirilmesi, uç birimlerde çalıştırılması ve cevapların toplanması işlemlerinin nasıl yapıldığı anlatılacaktır.

### 3.2.1 Veri Üzerinde Yapılan Ön İşlemler

Tasarlanan sisteme veri toplu şekilde tek seferde yüklenmektedir. Yüklenen veri çeşitli aşamalardan geçecektir. Ancak veri yükleme işleminden önce, sonraki adımlarda kullanılacak bazı işlemler tamamlanmalıdır.

Tasarlanan sistem N-Triple yapısındaki RDF dosyalarının depolanması ve sorgulanmasına olanak vermektedir. Sistem üzerinde deney yapılması sırasında gerçek veri değil bir üretici yazılım tarafından üretilmiş veri kullanılmıştır. Veri üretimini detayları Ek-A içinde verilmiştir. Daha önceki bölümlerde belirtilmiş olduğu gibi N-Triple yapısı dosyaların her satırda bir üçlü içermesini gerektirmektedir. Bir özne, yüklem veya nesne, yüklenen veri kümesi içinde birden fazla kere geçebilir. Bu bileşenlerin sayılması ve her bileşenden kaç tane bulunduğunu gösteren bir indeksin hazırlanması, sorgu safhasında ihtiyaç duyulan bir bilgidir. Bu bilgiye, sorguların sonuçlarının, ağ üzerinde ne kadar trafik yoğunluğu oluşturacağının hesabı sırasında ihtiyaç duyulmaktadır. Bu hesapların detayları, verilerin sorgulanmasının anlatıldığı bölümde verilecektir.

Bileşenlerin sayılması işlemi Hadoop kümesi üzerinde çalıştırılan bir Map-Reduce uygulaması ile hesaplanmaktadır. İlk olarak işlenecek verinin Hadoop içinden okunmasını sağlayan bir kod bloğu çalıştırılarak, RDF dosyaları Map-Reduce uygulamasına verilir. Uygulama, tüm üçlü bileşenlerin sayısını içeren bir çıktı dosyasını üretir ve bunu dosya sistemi üzerine kayıt eder. Bu işlem her toplu veri yüklenmesi işlemi için bir kere yapılmaktadır. Dolayısı ile sorgu sürelerine değil, sistemin kurulumu için gereken süreye etkisi bulunmaktadır. Bu durum sisteme daha sonradan veri eklenmesine engel değildir, ancak verinin parçalanması kümeleme algoritmalarına dayalı bir süreç olduğu için yeni verinin hangi uç birimde depolanması gerektiğinin tespiti güç olacaktır.

Bu safhaya farklı Map-Reduce uygulamaları ekleyerek veri üzerinde yapılmak istenen diğer ön işlemler de yapılabilir. Birçok benzer sistem bu safhada veri temizleme süreçleri uygulamakta, tekrar eden veya belli özellikleri sağlamayan verilerin, ana veri kümesinden çıkarılmasını sağlamaktadırlar. Örneğin, verinin parçalanması sırasında problem oluşturma ihtimali fazla olan, yüksek dereceli çizge düğümlerinin elenmesi istenilebilecek durumlardandır.

### 3.2.2 RDF Verilerin Dağıtık Ortamda Depolanması

İçinde bulunan tüm üçlü bileşenlerin sayılması işlemi tamamlanan RDF dosyaları, sonraki adımda, uç birimlere gönderilmek üzere METIS aracılığı ile parçalanır ve DRS uygulaması yardımı ile, ağ üzerinden, uç birimlerde bulunan Fuseki servisi aracılığı ile küme birimlerine yüklenir. Verinin uç birim üzerinde depolanmasının tamamlanmasıyla birlikte, sistem sorgulanmaya hazır hale gelecektir. Aşağıdaki bölümlerde verinin parçalanması ve dağıtılması süreçleri daha detaylı olarak anlatılmıştır.

#### 3.2.2.1 Verinin Parçalanması

Tasarlanan sistemin dağıtık yapıda olması dolayısı ile verilerin bu dağıtık sistem üzerinde ayrı uç birimlerde depolanması gerekmektedir. Bu noktada, verilerin uç

birimlere hangi şekilde dağıtılması gerektiği sorusu ortaya çıkmaktadır. Bunun için bir çok farklı yöntem uygulanabilir. Kullanılabilecek en temel yaklaşımlar,

1. sıralı hale getirilmiş üçlülerin, tüm uç birimlere, sıraya uygun bir şekilde eşit miktarda dağıtılması,
2. üçlünün özne bileşeni kullanılarak bir hash metoduna göre ayrılması [42, 37, 24],
3. verinin rastgele dağıtılması, ancak üçlü bileşenlerinden herhangi birine veya tamamına göre bir indeksinin oluşturulması,
4. verinin çizge özelliği olması göz önüne alınarak, çizge parçalama algoritmalarının kullanılması yolu ile parçalanması [28]

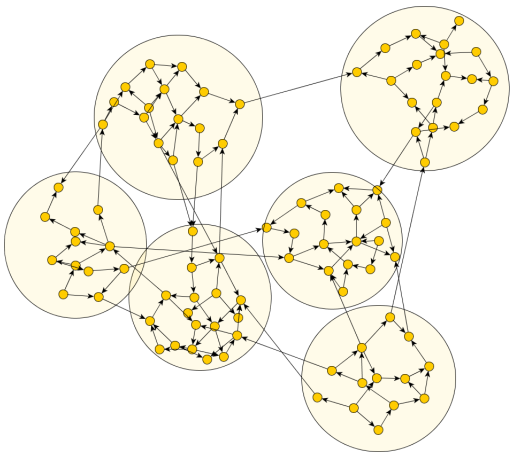
yöntemleridir. Bu yaklaşımların farklı avantajları ve dezavantajları bulunmaktadır. Birinci yöntemde veri sıralı olduğu ve eşit bölündüğü için, aranan bir üçlünün hangi uç birimde olduğunu bulmak çok basittir. İkinci yöntem, matematiksel bir yaklaşım ile bir hesaba dayanarak veriyi parçaladığı için, veriyi bulmak isteyen birim, aynı hesabı kullanarak aradığı verinin hangi uç birimde bulunması gerektiğinin hesabını yapabilir. Üçüncü yöntemde diğerlerine benzer şekilde, verinin hangi uç birimde olduğunun sorgusunun yapılabildiği bir indeks sağladığı için basit bir yöntemdir. Ancak bu yöntem verinin çok büyük olduğu durumlarda, indeks içinde arama süresi uzun olacağı için çok tercih edilebilir değildir. Birinci ve ikinci yöntemler, kolay kullanımı olan ve hızlı yöntemler olmalarına rağmen, verinin çizge yapısında oluşunu hiçbir şekilde göz önünde tutmamakta, bu yüzden de ilişkili çizge birimlerini içeren sorgulamalar sırasında, küme uç birimleri arasında yoğun haberleşme ihtiyacı ortaya çıkarmaktadır. Dördüncü yöntem ise verinin eşit dağıtılacağı garantisini vermeyen, ancak, komşuluk ve bağlantı sayısı gibi durumları göz önüne alarak veriyi parçalayan bir yaklaşımdır. Bu yöntemin diğer yöntemlere oranla önemli bir avantajı bulunmaktadır. Çizge içinde, birbirine yakın olan ve aralarında birçok kısa yol bulunan çizge düğümleri, parçalama işlemi sonucunda, büyük ihtimalle aynı parça içinde olacaklardır. Bu sayede, bu veri üzerinde yapılan bir sorgu için, verilecek

cevabın çok büyük bir kısmı tek uç birim tarafından cevaplanacaktır. Bu durum, sorguları toplayan uç birimlerin, veri birleştirme (Join) iş yüklerini çok büyük oranda azaltacaktır. Buna karşın, ilk üç yöntemde, bütün uç birimlerden cevap geleceği beklenebilir. Veri miktarının çok fazla olduğu durumlarda, her bir uç birimde bulunan parça büyüklüğü ve dolayısı ile dönecek olan cevap büyüklükleri fazla olacağından, verinin birleştirilmesi işlemini yapan birimin iş yükü artacaktır.

Birden fazla, N-Triple RDF dosyası halinde bulunan üretilmiş veri, büyük tek bir çizgeye karşılık gelmektedir. Çünkü, farklı dosyalarda bulunan bu üçlüler arasında bağlantılar bulunmaktadır. Çizge verilerin parçalanması konusu, akademik alanda çok yoğun olarak çalışılmış bir problemdir. Bu çalışmalar sonucunda elde edilen algoritmalar göz önüne alınarak geliştirilmiş ve çok yaygın olarak kullanılmakta olan bazı çizge parçalayıcı yazılımlar bulunmaktadır. Bu tez çalışmasında en çok kullanılan parçalayıcılardan birisi olan METIS [27] kullanılmıştır.

METIS, verilen bir çizgeyi, kullanıcının girdiği sayıda alt çizgeye bölen bir yazılımdır. Bölme işlemi için, kullanıcının tercih edebileceği iki farklı algoritma sunmaktadır. Bunlar, çok seviyeli özyineli bölme (multilevel recursive bisection) [31] ve çok seviyeli k-yol parçalama (multilevel k-way partitioning) [30] algoritmalarıdır. Bu tez çalışmasında, METIS tarafından varsayılan algoritma olarak sunulan, çok seviyeli k-yol algoritması kullanılmıştır. Çok seviyeli k-yol algoritması , METIS içinde gerçekleştirilen diğer algoritma olan, çok seviyeli özyineli bölme algoritmasından daha hızlı çalışmaktadır[30]. Ancak, veri parçalama işlemi, sorgu sürelerinin ölçülmesinden önce yapılan ayrı bir işlem olduğu için, sorgu cevaplama sürelerine doğrudan etkisi bulunmamaktadır. Şekil 3.8, bir çizgenin, çok seviyeli k-yol algoritması kullanılarak oluşturulan parçalarını göstermektedir. Şekilde dikkat edilmesi gereken nokta, ayrı parçalar halinde gruplanan çizge düğümleri arasındaki bağlantı sayılarının minimize edilmiş olmasıdır. Bu algoritma verilen çizgeyi, kullanıcı tarafından girilen sayıda alt kümeye ayırırken, farklı alt küme elemanları arasında, mümkün olan en az bağlantının kalmasını sağlayacak şekilde çalışır. Bu sayede, alt parçaların farklı küme uç birimleri üzerinde işlenmesi ya da sorgulanması sırasında, ihtiyaç

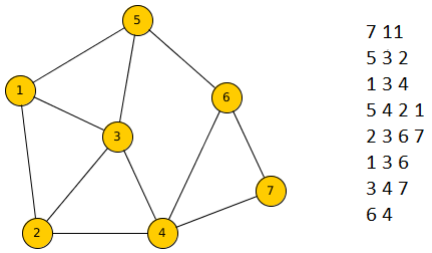
duyulabilecek olan, uç birimler arası haberleşme, en alt seviyeye indirgenmiş olmaktadır. METIS dışında kullanılabilecek diğer bir açık kaynak kodlu çizge parçalayıcı ise KaHIP [43] yazılımıdır. METIS ile çok benzer yeteneklere sahip olan KaHIP daha çok algoritma gerçekleştirimi içermektedir.



Şekil 3.8: METIS multilevel k-way algoritması ile parçalanan çizge

METIS programının, girdi olarak kabul ettiği, kendine özgü bir çizge notasyonu bulunmaktadır. Dolayısı ile, bu tez çalışması sırasında kullanılan RDF dosyaları içerisinde bulunan, N-Triple yapısındaki çizgenin, METIS çizge yapısına çevrilmesi gerekmektedir. DRS Merkez Uygulaması kapsamında, bu çevirimi iki yönlü olarak yapabilen, RDF parçalama ve dağıtma bileşeni isimli uygulama geliştirilmiştir. METIS komut satırı uygulamalarının, bu bileşen içerisinden kullanılabilmesi amacı ile gerekli kabuk betikleri yazılmıştır[10]. METIS formatına çevrilen çizgenin, geçerli bir METIS çizgesi olup olmadığı, METIS tarafından sağlanan "graphchk.exe" uygulaması aracılığı ile kontrol edilebilmektedir. METIS, parçalama işlemi sonucunda, hangi çizge biriminin hangi parçaya atandığını gösteren bir düz metin dosyası üretmektedir. Bu dosya içinde, bulunması gereken parça numarası belirtilmiş olan çizge bileşenleri, RDf üçlülerinde bulunan özne veya nesneden birisi olabilir. RDF parçalama ve dağıtma bileşeni, RDF dosyalarında bulunan üçlüleri özne değerlerine göre, bu çıktıyı kullanarak ayırmaktadır.

[10]METIS formatına çevrilen çizgenin, geçerli bir METIS çizgesi olup olmadığı, METIS tarafından sağlanan "graphchk.exe" uygulaması aracılığı ile kontrol edilebilmektedir.




Şekil 3.9: METIS Çizge Yapısı

Şekil 3.9'da verilmiş olan çizge için, METIS çizge dosyası içeriği yan tarafında verilmiştir. Buradaki ilk satır çizgedeki toplam düğüm sayısı ve bağlantı sayısını vermektedir. Diğer satırların hepsinde, satır numarası ile eş olan düğümün komşularının hangileri olduğu verilmiştir. Örneğin, 1. düğümün komşuları 2,3 ve 5'dir.

LUBM ile üretilmiş verinin parçalama süreci aşağıdaki adımlarla verilmiştir.

1. Tasarlanan sistem, N-Triple RDF dosyaları ile çalışacağı ve üretilmiş veri içinde birden çok RDF dosyası bulunmasından dolayı, veriler tek RDF dosyası altında toplanır. Bu işlemi gerçekleştirmek amacı ile basit bir kabuk betiği hazırlanmıştır.

2. Elde edilen RDF dosyası içinde bulunan çizge, RDF Parçalama ve Dağıtma Bileşeni isimli uygulama ile, tek bir METIS çizgesine çevrilir. METIS çizge dosyasında ilişkilere karşılık gelen üçlü bileşenleri (predicate) bulunmaz. METIS çizge yapısı, yönlü bir çizge yapısı değildir. Ancak RDF çizgeleri yönlü çizgelerdir. Dolayısı ile RDF çizgelerinde, iki çizge düğümü arasında, birden fazla ilişki aracılığı ile bağlantı olabilir. Bu durum METIS'in çalışması için bir problem teşkil etmemektedir. Çünkü METIS sadece komşuluk yapısını göz önüne almaktadır. Bu yüzden, çevirme işlemini yapan yazılım bileşeni, iki çizge düğümü arasında birden fazla bağlantı olması durumunda, bunlardan sadece birincisini kullanmaktadır.

3. Oluşturulan çizge dosyası, METIS tarafından okunarak, kullanıcının belirttiği sayıda parçaya, kullanıcının belirttiği algoritma(multilevel recursive

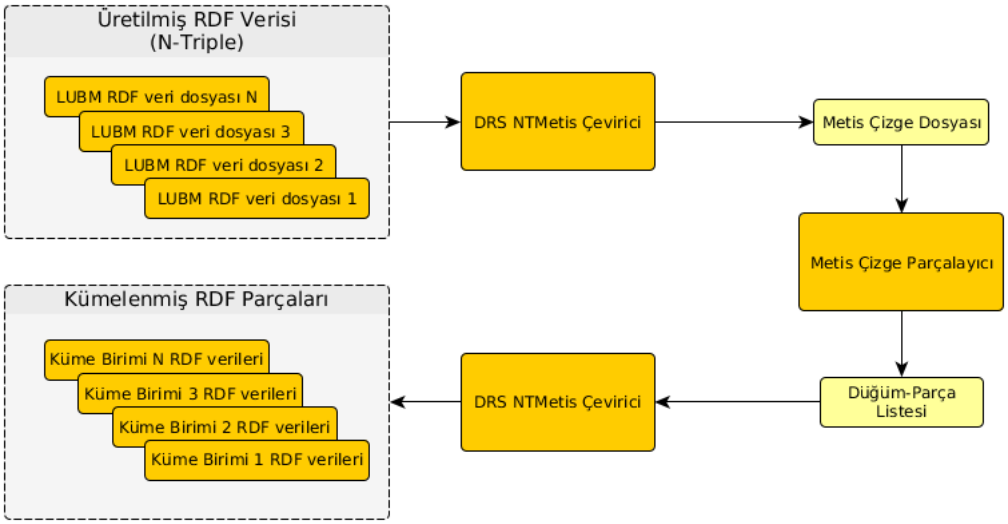
bisection [31] veya multilevel k-way partitioning [30]) ile bölünür. Bu tez çalışmasında multilevel k-way partitioning algoritması kullanılmıştır.

4. Bölüm işlemi sonucunda, hangi çizge düğümünün hangi parça içinde olması gerektiğini gösteren tek bir çıktı dosyası oluşturulur. Burada dikkat edilmesi gereken nokta, METIS'in üretmiş olduğu çıktı dosyasında, üçlülerin değil, sadece çizge düğümlerinin hangi parçalarda olması gerektiğinin bulunuyor olmasıdır. Ancak, veri parçalama işleminin asıl hedefi, RDF dosyalarındaki üçlülerin, alt çizgelere nasıl ayrılacağı sorusuna cevap vermektir. Dolayısı ile, bir üçlüye ait özne ve nesne, METIS tarafından ayrı alt çizgelere atanmış olabilir. Bu durumda, çıktı dosyası içindeki çizge düğümü ile ilgili üçlünün, RDF dosyalarının parçalanması işlemi sırasında, hangi parçaya atanacağı belirsiz kalmaktadır. Bu durumu çözmek için izlenecek yol, her üçlünün, o üçlüye ait öznenin, METIS çıktısı içinde atandığı alt çizgeye atanmasıdır. Diğer alternatif olan, üçlünün, nesnenin atandığı RDF parçasına atanması durumu eşdeğer simetrik çözümdür. Burada, tercih olarak özne seçilmiştir.

5. Madde 4'de anlatılan sebepten ötürü, veri kümesi içindeki her üçlü, tek tek okunur. Okunan üçlünün özne bileşeni tespit edilir. Belirlenen özne değeri, METIS çıktısı içinde aranarak, hangi numaralı alt çizgeye atandığı bulunur. Ele alınmış olan üçlü, numarası tespit edilen çizge parçasını içeren RDF dosyasına kopyalanır.

6. İşlem tüm üçlüler için tekrarlandığında, başlangıçtaki RDF dosyaları, istenen parça sayısında (bu sayı kümedeki uç birim bilgisayar sayısına eşittir) yeni RDF dosyaları oluşturulmuş olur.

   Şekil 3.10'de veri parçalama süreci gösterilmiştir. Şekilde gösterilen DRS NTMetis Çevirici birimi, DRS merkez uygulamasının, RDF parçalama ve dağıtma bileşeni içinde yer almaktadır.



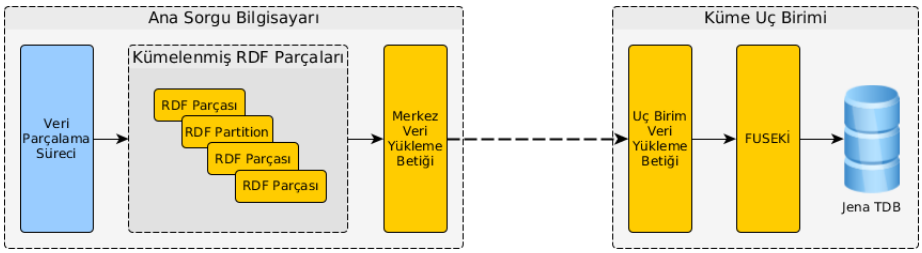

Şekil 3.10: METIS İle Çizge Parçalama

#### 3.2.2.2 Verinin Dağıtılması

Verilerin Parçalanması bölümünde açıklanan süreç sonucunda, kümeleme işlemi ile parçalanmış üçlülerden oluşan RDF dosyaları elde edilmiştir. Bu dosyalar, küme uç birimleri üzerinde bulunan ve Jena Fuseki aracılığı ile erişilebilen, Jena TDB üçlü veritabanlarına gönderilir. Bu amaçla, ilk olarak dosyalar, DRS Merkez uygulaması altında çalışan betikler kullanılarak, tüm uç birimler tarafından erişilebilen bir NFS (Network File System) dizinine kaydedilir. Uç birim üzerinde bulunan ve uzaktan çalıştırılabilen diğer bir betiğin, DRS Merkez Uygulaması tarafından yürütülmesi ile, paylaşımlı dosya sisteminden okunan RDF dosyasının içeriği, Jena TDB üçlü veritabanına kaydedilir. Verinin Jena TDB'ye yüklenmesi işlemi için, Jena TDB tarafından sağlanan komut satırı uygulamaları kullanılır. Parçalanmış RDF verilerinin hangi bilgisayarlarda depolandığının, yani hangi IP adresinde verinin hangi parçasının durduğunun bir önemi bulunmamaktadır. Tasarlanan sistem bu açıdan tam olarak simetrik bir yapıya sahiptir.

DRS merkez uygulaması kapsamında, verilerin uç birimlerde bulunan Jena TDB veritabanlarından temizlenmesi ve yeniden yüklenmesi için gereken ayrı kabuk betikleri de yazılmıştır. Bu işlemler, veri yükleme sürecinin bir parçası

oldukları için, sorgu cevaplama sürelerinin hesabına bir etkisi bulunmamaktadır. Şekil 3.11'da veri yükleme süreci gösterilmiştir.




Şekil 3.11: Verinin uç birimlere dağıtılması

### 3.2.3 RDF Verilerinin Dağıtık Ortamda Sorgulanması

Bu bölümde, çizge yapısındaki verilerin dağıtık ortamda sorgulanmasını zorlaştıran durumlara değinilecek ve buna karşı tez çalışmasında uygulanmış olan yaklaşımın detayları anlatılacaktır.

Verilerin METIS ile çok seviyeli k-yol algoritması ile parçalara ayrılarak, küme uç birimleri üzerinde çalışan ve Jena Fuseki aracılığı ile erişilebilen, Jena TDB üçlü veritabanları üzerinde depolanması sonrasında, her uç birim, kendi depoladığı veri ile ilgili sorgulara cevap verebilir duruma gelmiştir. Ancak, bu noktadan itibaren kullanıcı tarafından girilebilecek herhangi bir sorgunun tek bir noktadan cevaplanması mümkün olmadığı için, cevapların oluşturulması amacı ile fazladan işlemler yapılması gerekmektedir.

#### 3.2.3.1 RDF Verinin Dağıtık Ortamda Saklanmasının Sorgulamaya Getirdiği Zorluk ve Çözüm Yaklaşımı

Çizge veriler, temelinde birçok çizge düğümü ve onları birleştiren kenarlardan oluşmaktadır. Bu tür bir verinin, birçok farklı yapı içinde depolanması mümkündür. Ancak, tez çalışmasında kullanılan N-Triple yapısı ile, daha önce açıklandığı gibi, her biri bir satırda bulunan üçlüler şeklinde ifade edilirler. Bir

SPARQL sorgusunun cevaplanması işlemi ise, bu çizge üzerinde bulunan bir yol kalıbını sağlayan düğümlerin bulunmasıdır. Burada bahsedilen yol, esasında, üzerinde sorgu çalıştırılan çizgeye ait bir alt çizgedir. Örneğin aşağıdaki SPARQL sorgusu, "GraduateCourse0" türündeki dersleri ve bu dersleri alan öğrencilerin listesini sorgulamaktadır.

SPARQL 3.1: SPARQL Sorgusu

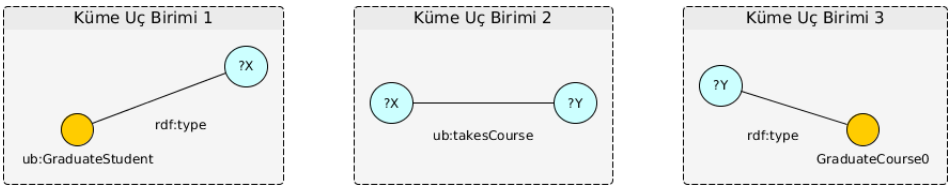
```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
3 SELECT ?X ?Y
4 WHERE
5 {
6   ?X rdf:type ub:GraduateStudent .
7   ?Y rdf:type <http://www.Department0.University0.edu/GraduateCourse0> .
8   ?X ub:takesCourse ?Y
9 }
```

Bu sorgu, çizge içinde Şekil 3.12'de gösterilen yolu yada alt çizgeyi sağlayan tüm düğüm değerlerini istemektedir. Dolayısı ile bu sorguya verilen cevaplar, belirtilen yol kalıbını, aynı anda birlikte sağlayan tüm X ve Y değişkenleridir. Doğal olarak, geri alınan cevap boş küme olmadığı sürece, eşit sayıda X ve Y değeri içerecektir. Ancak SPARQL 3.1 sorgusu içinde geçen "?X ub:takesCourse ?Y" ifadesi sorgudan çıkarılır ise, cevap olarak dönen X ve Y değerlerinin sayısı farklı olabilir.


Şekil 3.12: SPARQL sorgusu tarafından sağlanması gereken alt çizge

Sorgu içinde istenen her bir üçlü, çizge üzerinde sağlanması gereken bir üçlüye karşılık gelmektedir. Dağıtık sorgulama için en önemli problem, çizgeye ait her bir üçlünün farklı bir uç birimde depolanmış olma ihtimalinin olmasıdır. Örneğin,

SPARQL 3.1 sorgusu ile tarif edilen alt çizge veya yol, uç birimlere Şekil 3.13'deki gibi dağılmış olabilir.




Şekil 3.13: SPARQL sorgusu tarafından tarif edilen alt çizgeye ait parçaların olası dağılımlarından bir tanesi

Ancak bu durum, kullanıcı tarafından verilen sorgunun, ilk hali ile cevaplanmasını imkansız hale getirmektedir. Bu sebeple, kullanıcının girdiği sorgunun, uç birimlere gönderilmeden önce işlenerek, verinin yeni haline uygun bir şekilde yeniden yapılandırılması gerekmektedir. Doğal olarak, verinin dağıtık halde bulunuyor olmasından dolayı aynı sorgu ile sorgulanamıyor olmasına rağmen, yeni oluşturulacak sorguların da sonuç olarak orijinal sorgu ile aynı sonuçları veriyor olması gerekmektedir. Veri parçalama işlemi sırasında herhangi bir indeksleme işlemi yapılmadığı için, verinin hangi parçasının, hangi uç birimde depolandığına dair bir bilgi bulunmamaktadır. Dolayısı ile sorgulama için uygulanacak yöntem tüm uç birimler üzerinde çalıştırılabilecek şekilde simetrik olmalıdır. Örneğin Şekil 3.13'de verilen biçimde dağıtılmış bir yapıda, ayrı ayrı orijinal sorgunun çalıştırılması, üç küme uç biriminden de boş cevap kümelerinin alınmasına yol açacaktır. Ancak, orijinal sorgunun SPARQL 3.2 içinde gösterilen sorgular halinde bölünmesi ve sırası ile küme uç birimi 1, küme uç birimi 2 ve küme uç birimi 3'e sorulması durumunda, elde edilen değerler, orijinal sorgunun vereceği cevabı içeren daha büyük bir veri kümesi oluşturacaktır. Örneğin, S2 sorgusu için küme uç birimi 2'den elde edilen X değerlerinin ancak bir kısmı, S1 sorgusunun küme uç birimi 1'de çalıştırılması ile elde edilecek olan X değerlerine eşit olacaktır. Benzer şekilde S2 sorgusunun küme uç birimi 2 üzerinde çalıştırılması ile elde edilecek olan Y değerlerinin ancak bir kısmı, S3 sorgusunun küme uç birimi 3 üzerinde çalıştırılması ile elde edilecek olan Y değerlerine eşit olacaktır. Dolayısı ile sadece, sorgunun veriye uygun biçimde parçalanması yeterli olmamakta, elde edilen sonuçların doğru biçimde birleştirilmesi de gerekmektedir. Doğru sonuçları

elde etmek için bu üç ayrı küme uç biriminden elde edilen X ve Y değerlerinin kendi aralarında kesişim kümesi alınmalıdır. Bu şekilde üç sorguyu da sağlayan X ve Y değerleri bulunmuş olacaktır.

SPARQL 3.2: SPARQL 3.1 sorgusundan üretilen SELECT sorguları

```
S1
-----------------------------------------------------------------------
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{
  ?X rdf:type ub:GraduateStudent .
}

S2
-------------------------------------------------------------------------
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y
WHERE
{
  ?X ub:takesCourse ?Y
}


S3
-----------------------------------------------------------------------
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?Y
WHERE
{
  ?Y rdf:type <http://www.Department0.University0.edu/GraduateCourse0> .
}
```

#### 3.2.3.2 SPARQL Sorgularının Parçalanması

Verinin dağıtık bir ortamda depolanması dolayısı ile oluşan sorgulama problemi önceki bölümde açıklanmıştır. Bahsedilen problemin çözümü, sorguların daha küçük sorgulara dönüştürülerek, cevapların ayrı noktalardan elde edilmesi ve daha sonra bu cevapların birleştirilerek, kullanıcının sorgusuna cevap oluşturulmasıdır.

Sorguların parçalanması, verilerin parçalanmasının doğal bir sonucudur. Veri parçalama işlemi sırasında, farklı RDF dosyaları arasında dağıtılan, daha küçük parçalara ayrılmayan en küçük veri yapısı, yani atomik seviyedeki veri yapısı üçlülerdir. Sorgular içinde sağlanması gereken ve üçlüler şeklinde ifade edilen yolların her biri, veri içindeki bu üçlülere karşılık geldiği için, sorgularında bulunan üçlüler, yani sağlanması gereken yollar baz alınacak şekilde parçalama yapılmalıdır.

Sorgu parçalama işlemi, DRS merkez uygulaması, sorgu parçalama bileşeni tarafından yapılmaktadır. Bu bileşen, kullanıcı tarafından verilen bir RDF sorgusunun parçalarını düz metin formatında geri dönen, açık kaynak kodlu, Rasqal Roqet komut satırı uygulamasını kullanmıştır[11]. Dönen metin içinden Triple alanları değişkenlere atanmak sureti ile uygulama içinde kullanılmıştır. DRS Merkez uygulaması, elde edilen üçlüleri kullanarak farklı amaçlara yönelik sorgular oluşturabilmekte ve bu sorguları uç birimler üzerinde çalıştırılmak üzere, bir yığıt veri yapısı içinde, küme uç birimlerine göndermektedir. Ayrı sorgu türlerine ihtiyaç duyulmasının sebebi, bazı sorguların sadece çizge düğüm değerleri almaya yönelik olmasına karşın, diğerlerinin sadece verinin ilgili uç birimde var olup olmadığının tespiti için kullanılacak olmasıdır. Yeni sorguların oluşturulması ve çalıştırılması Jena uygulama geliştirme çatısı kullanılarak, DRS Merkez Uygulaması içinde gerçekleştirilmiştir. Rasqal Roqet komut satırı uygulaması kullanılarak SPARQL 3.1'in işlenmesi sonucunda SPARQL 3.3 içinde gösterilen çıktı elde edilmiştir. Bu çıktı sorguya ait tüm alanları ayrı ayrı göstermektedir.

SPARQL 3.3: Rasqal Roqet komut satırı uygulaması ile işlenen SPARQL 3.1'in çıktısı

[11] http://librdf.org/rasqal/

```
query verb: SELECT
query bound variables (2): X, Y
query Basic graph pattern[0]
{
        triples {
                triple #0 { triple(variable(X), rdf:type,
                    ub:GraduateStudent) }
                triple #1 { triple(variable(Y), rdf:type,
                    <http://www.Department0.University0.edu/GraduateCourse0>)
                    }
                triple #2 { triple(variable(X),ub:takesCourse,
                    variable(Y) ) }
        }
}
```

Sorgu parçalama işleminin ilk safhasında, yeni sorgularda kullanılacak olan üçlüler orijinal sorgudan elde edilir. Bunlar Rasqal Roqet çıktısında "triples" ile gösterilen alanda numaralanmış olarak bulunan "triple" değerleridir. Elde edilen üçlülerin her biri, yeni oluşturulacak sorgulardan sadece birinde kullanılacak şekilde, ihtiyaca uygun soru yüklemine göre [12], yeni sorgular elde edilir. Örneğin önceki bölümde verilen SPARQL 3.1 sorgusunun ASK sorgularına çevrilmesi durumunda SPARQL 3.4 içinde verilen alt sorgular elde edilir. Burada kullanılan, DRS merkez uygulaması, alt sorgu oluşturma bileşeni, kullanılan üçlü içinde geçen değişken değerine uygun olarak sorgu değişkenleri üretmektedir. Dolayısı ile bazı sorgular sadece X ve Y değişkenlerini sorgulayacak şekilde üretilmişken, bazıları iki değişkeni de sorgulayacak şekilde oluşturulmuştur.

SPARQL 3.4: SPARQL sorgusundan üretilen ASK sorguları

```

ASKQ1
-------------------------------------------------------------------------
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

[12] ASK veya SELECT sorgu yüklemleri

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
ASK ?X
WHERE
{
  ?X rdf:type ub:GraduateStudent .
}

ASKQ2
-------------------------------------------------------------------------
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
ASK ?X ?Y
WHERE
{
  ?X ub:takesCourse ?Y
}

ASK Q3
-------------------------------------------------------------------------
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
ASK ?Y
WHERE
{
  ?Y rdf:type <http://www.Department0.University0.edu/GraduateCourse0> .
}
```

ASK sorgularının oluşturulması ile aynı yöntem kullanılarak, uç birimler üzerinde çalıştırılacak olan asıl alt SELECT sorguları da hazırlanabilmektedir. İşlemdeki tek fark, sorgu yüklemi parametresinin ASK yerine SELECT olarak girilmiş olmasıdır. Oluşan SELECT sorguları SPARQL 3.2'de gösterilmiştir.

Oluşturulan ASK sorguları küme içindeki tüm uç birimlerde çalıştırılırken,

SELECT sorguları sadece cevap verebilecek uç birimlere bir yığıt içinde gönderilmektedirler.

#### 3.2.3.3 Alt SPARQL Sorgularının Yük Değerlerinin Hesaplanması

Bu tez çalışmasında, sorguların yük değeri, sorgunun cevaplanması durumunda oluşan cevap büyüklüğünü tanımlamak için kullanılmıştır. Kullanılan yöntem oluşacak olan trafik yoğunluğunu göreceli bir şekilde hesaplayacaktır. Bir sorguya verilen cevap, eğer çok fazla sayıda eleman içeren bir sonuç ise, bu cevabın bilgisayarlar arasında aktarılması o kadar çok zaman alacaktır. Dolayısı ile, orijinal sorgunun parçalanmasından sonra oluşturulan alt sorguların, dağıtık olarak sorgulanması öncesinde, hangi alt sorgunun ağda ne kadar trafik yoğunluğu oluşturacağının tespit edilmesi, sorguların uygun sırada çalıştırılması ve ağ kullanımını azaltılması açısından önemlidir.

Alt sorguların yük değerlerinin hesaplanması sırasında aşağıdaki adımlar izlenir.

1. Alt sorgu üçlüsü bileşenlerine (özne, yüklem ve nesne) ayrılır.
2. Bileşenlerden değişken olmayanlar seçilir.
3. Değişken olmayan bileşenlerin, RDF dosyasında kaç defa kullanıldığı tespit etmek için, Hadoop tarafından oluşturulmuş olan indeks kullanılır. Aranan bileşen indeksten bulunarak karşılığı olan değer okunur.
4. Değişken olmayan bileşenlerden, RDF dosyasında en az sayıda bulunan, değişken bileşen için bir üst sınır olacağından, değişkenin, dolayısı ile sorgunun yük değeri belirlenmiş olur.

Örnek olarak, yukarıda verilen adımların, SPARQL 3.1 sorgusuna ait alt sorgulardan birisi olan SPARQL 3.5 sorgusu için uygulanışı aşağıda maddeler halinde verilmiştir.

SPARQL 3.5: Örnek SELECT sorgusu

```
1
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
4 SELECT ?X
5 WHERE
6 {
7   ?X rdf:type ub:GraduateStudent .
8 }
```

1. Alt sorgu üçlüsünün bileşenlerine (özne, yüklem ve nesne) ayrılması ile "?X", "rdf:type" ve "ub:GraduateStudent" bileşenleri elde edilir.

2. Elde edilen üç bileşenden "?X" bir değişken olduğu için, ele alınmaz. Değişken adı olmadıkları için "rdf:type" ve "ub:GraduateStudent" bileşenleri seçilir.

3. Değişken olmayan bu bileşenler, Hadoop tarafından oluşturulan indeks dosyası içinde arandıkları zaman, "rdf:type" için 20674 ve "ub:GraduateStudent" için 1874 değerlerini vermektedir.

4. Dolayısı ile örnek olarak verilen üçlüyü sağlayan ?X değişken sayısı en fazla 1874 olabilir. En fazla ibaresinin kullanılmasının sebebi, RDF dosyasında geçen tüm "ub:GraduateStudent" ifadeleri, bu üçlüyü sağlayan ?X değerleri ile birlikte kullanılmış olmayabilirler. RDF dosyaları içinde farklı üçlüler içinde "ub:GraduateStudent" kullanılmış olabilir. Ancak eğer tüm "ub:GraduateStudent" bileşenleri bu sorgu içinde belirtilen üçlü içinde geçiyor olsaydı, "?X" bileşenin de kesin olarak 1874 defa kullanılmış olması gerektiği söylenebilirdi. Dolayısı ile burada, bu sorgudan dönmesi teorik olarak mümkün olan en büyük sayı tespit edilmektedir, ancak gerçek değer sayısı 1874'den küçük olabilir.

Bu işlem başlıkta da belirtildiği üzere sadece alt sorguların yüklerini hesaplamak için kullanılmaktadır. Bu sebepten ötürü, yük hesabına, birden fazla üçlüsü olan bir alt sorgunun hiçbir zaman girmeyeceği öngörülmüştür. Bu işlem sonucunda

elde edilen 1874 değeri, sorgunun bir küme uç birimine yapılması durumunda alınacak cevap içinde bulunabilecek "?X" değer sayısının alabileceği en yüksek değeri gösterdiği için, farklı sorguların birbirleri ile karşılaştırılması ile, ağda oluşturacakları trafik yoğunlukları göreceli olarak kıyaslanabilmektedir. Bu karşılaştırma, sorguların hangilerinin diğerlerinden önce yapılması gerektiğini belirlemede önemlidir. Çünkü, sorguların dağıtık ortamda yürütülme sürecinin anlatıldığı, 3.2.3.5 bölümünde bahsedileceği gibi, sorgu yığıtı içinde en sonda bulunan sorguların cevapları, küme uç birimleri arasında en fazla sayıda iletilecek olan paketlerdir. Dolayısı ile bu paketlerin boyutlarının küçük olması, genel olarak performansa artı yönde katkı sağlayacaktır.

#### 3.2.3.4 Dağıtık Sistem Alt Sorgu - IP Matrisinin Oluşturulması

Sistemde bulunan küme uç birimlerin, hangi alt sorgulara cevap verebileceğinin bilinmesi, sorgunun, cevap veremeyecek uç birimlere gereksiz bir şekilde gönderilmesini engelleyebilecektir. Bu şekilde, sorguların gereksiz ağ yoğunluğuna sebep olması ve cevaplayamayacakları sorguları alan küme uç birimlerinin, yerel veritabanlarında gereksiz aramalar yapması engellenebilecektir. Bunlar performans artışı sağlayabilecek önlemlerdir. Küme uç birimlerinin hangi sorgulara cevap verebileceğinin tespiti için, orijinal kullanıcı sorgusunda bulunan üçlü yollarının, küme uç birimlerine, ayrı ayrı sorulması yeterlidir. Bu amaçla, kullanıcı sorgusundaki her bir üçlü için, DRS merkez uygulamasına ait alt sorgu oluşturma bileşeni tarafından, ayrı bir ASK sorgusu hazırlanır ve bu sorgu tüm küme uç birimlerinde çalıştırılır. Eğer, küme uç birimi, sorgulanan üçlüyü içeriyorsa, ASK sorgusuna True, içermiyorsa False dönecektir. Üçlünün küme uç birimi üzerinde aranması sırasında, üçlüyü sağlayan değerlerden en az bir tanesinin bulunması durumunda, arama işlemi sonlandırılır. Bu sayede gereksiz sistem yoğunluğuna neden olunmaz. Bu bilgi kullanılarak aşağıdaki gibi bir matris oluşturulabilir. Bu matris içinde, N ile başlayan ifadeler (N1, N2, vb.) küme uç birimini, "ASKQ" ile başlayan ifadeler ise (ASKQ1, ASKQ2, vb.) ASK sorgularını göstermektedir. Değer olarak "1", ilgili birimin sorguyu cevaplayabileceğini, "0" ise cevaplayamayacağı gösterir.

Çizelge 3.3: Alt Sorgu - Uç Birim Matrisi

| Alt Sorgu/Küme Uç Birimi | N1 | N2 | N3 | N4 | N5 |
|---|---|---|---|---|---|
| ASKQ1 | 1 | 0 | 1 | 0 | 1 |
| ASKQ2 | 0 | 1 | 0 | 0 | 0 |
| ASKQ3 | 0 | 1 | 0 | 1 | 0 |

Bu matris, alt sorguların yığıt olarak dağıtımı sırasında, ilgili olmayan küme uç birimlerine yığıtın gönderilmesini engellemek için kullanıldığı gibi, bir alt sorgu için tüm uç birim değerlerinin sıfır olması durumunda, orijinal sorgunun tamamının cevaplanamayacağının tespit edilmesinde de kullanılmaktadır. Örneğin, ASKQ1 sorgusu için N1-N5 arasındaki tüm değerlerin sıfır olması durumunda, sistemin ASKQ1 alt sorgusuna cevap veremeyeceği anlamı çıkarılabilmektedir. ASKQ1, kullanıcının girmiş olduğu sorgudaki üçlü yollarından bir tanesine denk geldiği için, tüm değerlerin sıfır olması, tüm RDF verisi göze önüne alındığında, bu üçlüyü sağlayan bir veri bulunmadığı anlamına gelir. Bu durumda, diğer sorguların yapılmasına gerek kalmaz, çünkü birleştirme sırasında kullanılacak olan kümelerden bir tanesi boş küme olarak tespit edilmiştir.

#### 3.2.3.5 Sorguların Dağıtık Ortamda Yürütülme Süreçleri

Bu kısımda, kullanıcı tarafından verilen bir sorgunun, tasarlanan sistem içerisinde geçtiği süreçler, bir akış listesi olarak verilecek ve bazı adımların detayları alt başlıklar altında açıklanacaktır. Yürütülen adımlar, SPARQL 3.1 örneğindeki sorgu kullanılarak açıklanacaktır.

1. Kullanıcı, DRS merkez uygulamasına ait olan komut satırı bileşenini kullanarak, çalıştırmak istediği sorgu dosyasını girer (Ör. SPARQL 3.1).
2. DRS merkez uygulaması, sorgu parçalama bileşeni aracılığı ile, Rasqal Roqet komut satırı uygulamasını kullanarak sorguyu, bileşenlerine ayırır (Bkz. SPARQL Sorgularının Parçalanması bölümü, SPARQL 3.3).
3. SPARQL 3.3 içinde gösterilen her bir üçlü için, DRS merkez uygulaması alt sorgu oluşturma bileşeni tarafından bir ASK sorgusu oluşturulur (Bkz.

SPARQL Sorgularının Parçalanması bölümü, SPARQL 3.4).

4. DRS Merkez Uygulaması, konfigürasyon dosyası içinde tanımlı olan ve küme uç birimlerine ait IP adreslerini okur.

5. Oluşturulan tüm ASK sorguları, DRS merkez uygulaması, Alt Sorgu- IP matrisi oluşturma bileşeni tarafından, tüm IP adresleri üzerinde çalıştırılır ve daha sonraki adımlarda kullanılacak olan Alt Sorgu-IP matrisini oluşturur. Bu sayede hangi IP'nin, hangi sorgulara cevap verebileceği tespit edilmiş olur. Örnek bir alt sorgu-IP matrisi Çizelge 3.3 içinde gösterilmiştir.(Bkz. Dağıtık Sistem Alt Sorgu-IP Matrisinin Oluşturulması bölümü).

6. Madde 2'de oluşturulan üçlüler, DRS merkez uygulaması, alt sorgu oluşturma bileşeni tarafından tekrar kullanılarak, her bir üçlü için bir SELECT sorgusu olacak şekilde, alt sorgular oluşturulur. Oluşturulan SELECT sorguları SPARQL 3.2 içinde verilmiştir.

7. Madde 6'de oluşturulan SELECT sorguları için, DRS merkez uygulaması, alt sorgu yük hesaplama bileşeni tarafından yük hesabı yapılır (Bkz. Alt SPARQL Sorgularının Yük Değerlerinin Hesaplanması Bölümü). Bu adım sonunda hangi alt sorgunun, en fazla ne kadar ağ yoğunluğu oluşturabileceği belirlenmiş olur. Farklı alt sorgular için belirlenen bu değerler kullanılarak, alt sorgular yük değerlerine göre azalan sıraya dizilir. Burada, örnek için kullanılan SPARQL 3.1'de verilen sorguya ait olan ve SPARQL 3.2 içinde verilen alt sorguların yük sıralaması [S2, S1, S3] şeklindedir. Yani yük değeri en yüksek olan alt sorgu S2 ve en düşük olan S3'dür. Bu sıralama sorgu yığıtı oluşturulurken kullanılacak olan sıralamadır.

8. Madde-7 içinde oluşturulan alt sorgu yük sıralamasından en yüksek değere sahip olan alt sorgu listeden alınır ve bu sorguyu cevaplayabilen küme uç birim IP değerleri, Çizelge-3.3 içinde verilen alt sorgu-IP matrisinden belirlenir. Bu madde ve madde-9, tüm alt sorgular için tekrarlanacaktır. Dolayısı ile, ele alınan sorgu ve IP'ler, ilk turda S3 ve bu sorguyu cevaplayabilen küme uç birimleri IP2 ve IP4 iken, ikinci turda S1 sorgusu

ve IP2, üçüncü turda ise S2 sorgusu ve bu sorguyu cevaplayabilen IP1, IP3 ve IP5 küme uç birimleri olacaktır.

9. Her turda, ele alınan sorgu ve bu sorguyu cevaplayabilen IP'ler, sorgu-IP ikilileri halinde sorgu yığıtına eklenir. Dolayısı ile ilk turun sonunda yığıta, S3-IP2 ve S3-IP4 değerleri eklenecek ve yığıt [S3-IP4, S3-IP2] halini alacaktır. İkinci turda yığıta S1-IP1, S1-IP3 ve S1-IP5 değerleri eklenecek ve yığıt [S1-IP5, S1-IP3, S1-IP1, S3-IP4, S3-IP2] halini alacaktır. Son turda S2-IP2 değeri eklenecek ve yığıt son hali olan [S2-IP2, S1-IP5, S1-IP3, S1-IP1, S3-IP4, S3-IP2] biçimini alacaktır. Bu maddenin gerçekleştirilmesi ile sorguları ve çalıştırılacakları IP'leri içeren yığıt hazırlanmış olur.

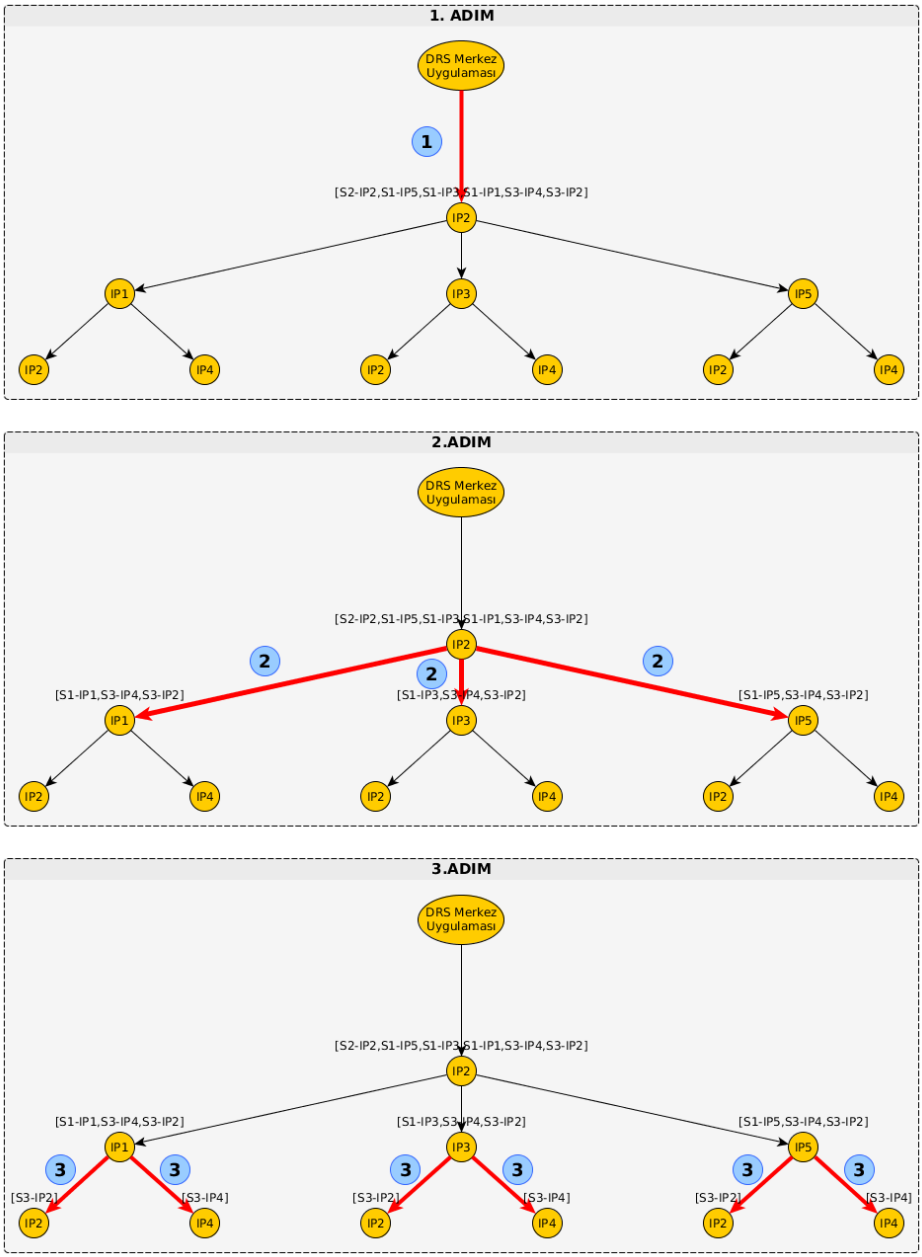
10. Madde-9 içinde hazırlanmış olan sorgu yığıtı, küme uç birimleri arasında devam edecek olan delegasyon ve cevaplama sürecine, DRS merkez uygulamasından başlar. DRS merkez uygulaması, sorgu yönetim bileşeni, yığıt içinde en üstte bulunan ve aynı sorgu ile oluşturulmuş yığıt elemanlarını tespit eder. Bu değer, verilen örnek için sadece S2-IP2'dir. Bu noktada birden fazla değer bulunma ihtimali vardır. Ancak bu örnekte, alt sorgu-IP matrisinde S2 sorgusu için sadece bir tek IP2 bulunduğundan, bu adımda yığıt içinden sadece S2-IP2 elemanı tespit edilebilmiştir. Elde edilen yığıt elemanının, IP değeri (IP2) bulunduktan sonra ilk delegasyon işlemi yapılarak, yığıt IP2 küme uç birimine gönderilir. Bu noktada sorgu yürütme işleminin devamı, IP2 üzerinde çalışan, DRS uç birim uygulamasının sorgu yürütme bileşenine devredilmiş olur. DRS merkez uygulaması sonuçların dönmesine kadar geçen süre içerisinde başka faaliyet yürütmeyecektir.

11. Sorgu yığıtını, DRS uç birim uygulamalarına ait sorgu yönetim bileşenleri aracılığı ile alan her bir uç birim( buradaki örnekte kendisine sorgu delege edilen tek küme uç birimi IP2'dir), sorgu yığıtı içinde bulunan ve kendi cevaplamaları gereken sorguyu, yerel Fuseki SPARQL uç birimine HTTP protokolü ile iletirler. Aynı zamanda, sorgu yönetme bileşeni, yığıt içinde bulunan en üst seviyedeki sorgu ile oluşturulmuş, yığıt elemanlarını yığıttan siler. Bunun sebebi bu sorguların cevabının alınmak üzere yerel Fuseki SPARQL uç birimine iletilmiş olmasıdır. Bu noktada sorgu yığıtı [S1-IP5,

S1-IP3, S1-IP1, S3-IP4, S3-IP2] haline dönüşmüştür. Yığıt üzerinde bu değişiklik yapıldıktan sonra, DRS uç birim uygulamasına ait sorgu delege bileşeni, yığıtta sırada bulunan en üst sorgu-IP elemanını (S1-IP5) okur . Okunan eleman içindeki, sorgu kısmı tespit edilir (S1). Yığıtta, bu sorgu ile eklenmiş başka eleman olup olmadığı kontrol edilir (Örnek yığıtta S1-IP5, S1-IP3, S1-IP1 elemanları bulunmaktadır). Eğer aynı sorguyu içeren başka sorgu-IP bileşenleri de bulunuyorsa, bu bileşenlerde ele alınarak, hepsinin IP kısımları bir listeye toplanır (IP5, IP3, IP1). Bu liste yığıtın son halinin delege edileceği küme uç birimleri listesini oluşturmaktadır. Sorgu delege bileşeni bu listeyi kullanarak, sorgunun geri kalan kısmını, o kısımları cevaplaması gereken diğer uç birimlere delege ederek, görevi diğer küme uç birimlerine devretmiş olur. Örnekte, IP5,IP3,IP1 uç birimlerine sırası ile [S1-IP5, S3-IP4, S3-IP2], [S1-IP3, S3-IP4, S3-IP2] ve [S1-IP1, S3-IP4, S3-IP2] yığıtları gönderilmiş olmaktadır. IP2 üzerindeki DRS uç birim uygulaması, sorguları gönderdikten sonra bu uç birimlerden cevap gelmesini bekler. Gelen cevaplar yerel Fuseki SPARQL uç birimi tarafından verilen cevaplar ile birleştirilecektir.

12. Özyineli bir şekilde, kendilerine gönderilen sorgu listesini alan IP5, IP3 ve IP1 uç birimleri Madde-10 ve Madde-11 içinde belirtilen süreçleri tekrarlayarak, kendi cevaplamaları gereken sorguları yığıttan çıkartarak geri kalan kısmı ilgili uç birimlere delege eder ve cevap beklerler.

13. Bu süreç, yığıt içinde delege edilebilecek başka eleman kalmayana kadar devam edecektir. Delege edilecek başka sorgu kalmadığında cevapların geri gönderilme süreci başlamaktadır. Bu sürecin yönü, sorgu yığıtının dağıtılma yönünün tersidir. Sorgu yığıtının, her hangi ir seviyede değiştirilmesini ve başka uç birimlere delegasyonunu yapmış olan tüm uç birimler, alttan gelecek olan cevapları beklemektedirler. Örnekte verilen sorgu yığıtı için oluşan delegasyon süreci Şekil 3.14'de görülmektedir. Sorgular kök birimden, uç birimlere doğru küçülen bir sorgu yığıtı halinde geldikten sonra, cevaplar en uç birimlerden ters yönde ilerlemektedir. Her uç birim, kendi cevapladığı ve delege ettiği sorgulara verilen cevapları birleştirmek sureti ile, kendi cevaplarını hazırlamış olmaktadırlar.



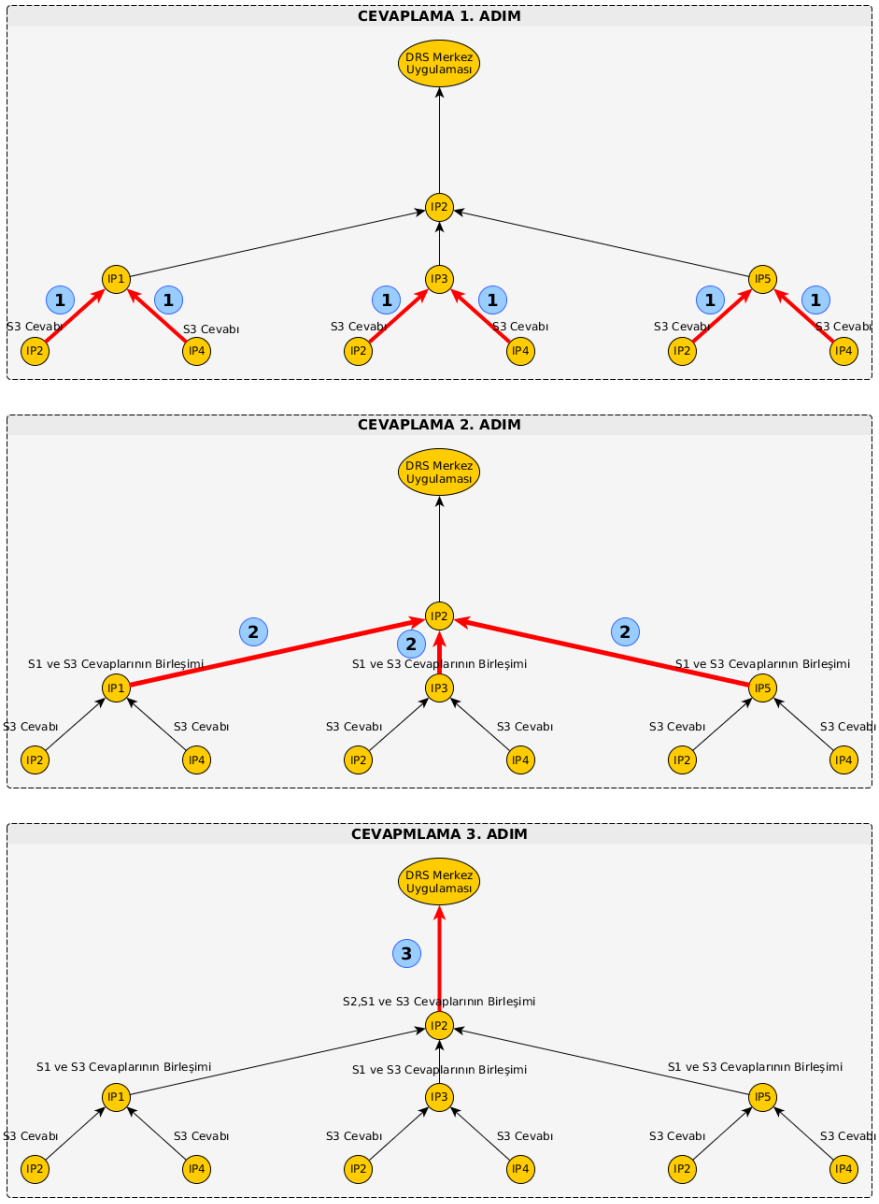

Şekil 3.14: SPARQL 3.1'e ait sorgu yığıtı delegasyon süreci

14. Cevaplama sürecinde, kendisine, sorgu yığıtının herhangi bir şekli delege edilen her küme uç birimi, cevabını, yığıtı kendisine delege eden küme birimine gönderir. Ancak, eğer bu küme uç birimi, yığıt üzerinde değişiklik yapmış ve başka küme uç birimlerine göndermiş ise, kendi cevabını göndermeden önce, delege ettiği sorgu yığıtına cevap verilmesini bekler. Kendi

göndereceği cevabı, delege ettiği yığıta gelen cevap ile, yerel Fuseki SPARQL uç biriminden aldığı cevapları, DRS uç birim uygulaması sorgu sonuçları birleştirme bileşeni aracılığı ile birleştirerek oluşturur. Burada bahsedilen birleştirme işlemi, iki ayrı kaynaktan gelen cevapların, ortak değişkenler üzerinden kesiştirilmesi işlemidir. Ortak olmayan değişkenler için cevap, değişken isimlerine göre ayrı kümelerin oluşturduğu listelerdir. Cevapların birleştirilmesi işleminin detayları, "Alt SPARQL Sorgu Sonuçlarının Birleştirilmesi" başlığı altında açıklanmıştır. Şekil 3.15 cevapların uç birimlerden geri gönderilmesi sürecini göstermektedir. Burada verilen yaklaşım birleştirme işlemlerinin dağıtık ve kademeli olarak yapılmasını sağladığından, birleştirme gibi yoğun işlem gerektiren bir adımın tek bilgisayar üzerinde yapılmasını engellemektedir. Ayrıca birleştirmenin tek seferde değil yeni cevaplar geldikçe yapılması da, anlık yük artışını minimal seviyede tutmaktadır.

Yukarıda, Madde 13'de bahsedilen yapıya ait olan Şekil 3.14 dikkatli bir şekilde incelendiği zaman, bazı sorguların, aynı uç birimlere farklı noktalardan ulaştığı görülmektedir. Bu durum performansı düşürücü bir durumdur. Ancak yapılması gerekli bir adımdır. Çünkü her birim, delege ettiği sorguların cevaplarını kendi yerel Fuseki SPARQL uç birimlerinden alınacak olan cevaplarla birleştirmek amacındadır. Bu yüzden, bir uç birimin daha önce cevaplamış ve sonuçlarını göndermiş olduğu bir sorgu, kendisine başka bir uç birim tarafından tekrar gönderilebilmektedir. Burada mantıksal bir hata bulunmamaktadır. Sorgunun farklı noktalardan geliyor olması, geri gönderilen cevapların farklı veri kümeleri ile birleştireceğini göstermektedir. Ancak aynı sorgunun tekrar cevaplanması, sorguyu cevaplayan uç birim üzerinde çalışan Fuseki SPARQL uç birimi ve ona bağlı Jena TDB için gereksiz bir zaman kaybı ve yük oluşturmaktadır. Bu süreç içinde gelebilecek yeni sorguların cevaplanması için kaynakların en üst seviyede serbest kalması önemlidir. Bu amaçla, "Sistemin Mimari Yapısı" bölümünde bahsedilmiş olan, Varnish HTTP Önbellek Vekil Sunucusu kullanılmaktadır. Bu sunucu, kurulu olduğu uç birime gönderilen sorgu yığıtlarını ve bunlara verilen cevapları önbellekte tutmaktadır. Bu sayede, kaynağı neresi olursa olsun, bir sorgu yığıtı tekrar gönderildiğinde, eğer önbellekte ilgili sorgu yığıtı ve cevabına

ait kayıt var ise, cevap önbellekten gönderilir. Ancak, eğer böyle bir kayıt yok ise, sorgu yığıtı Madde-10 ve Madde-11 anlatılan adımlar ile oluşturulur. Oluşturulan cevap Varnish tarafından tekrar önbelleğe alınır. Bu süreç yoğun sorgu yapılan bir sistemde, ağ yoğunluğunu düşürmez, ancak sorguların daha hızlı cevaplanmasını sağlar.




Şekil 3.15: Şekil 3.14 için Cevapların Dönüşü

Sistemin sorguları parçaladıktan sonra, yığıtlar halinde dağıtmasını ve uç birimlerin aldıkları sorgu yığıtlarını çalıştırmasını sağlayan algoritmalar Algoritma 1 ve Algoritma 2 içinde verilmiştir.

**Algoritma 1** SPARQL Sorgularının Çalıştırılması

**Input:** A SPARQL query $S$, $conf$ is the configuration file for DRS application
**Output:** A hashmap of query answers *Results*

```
1:
2:  IPList[] ← conf.getIPList()                    ▷ IPList is the array of IP's of the cluster nodes.
3:  queryComp[] ← rasqalroqet.parseQuery(S)        ▷ queryComp is the hashmap object that holds query components
4:  triples[] ← queryComp[triples]                 ▷ triples is the array that holds query triples
5:
6:  /* Create Ask and Select sub queries using triples array */
7:  for all triple in triples[] do
8:      /* askQueries is the array that holds ASK sub queries */
9:      askQueries[i] ← createAskQ(triple)
10:     /* selectQueries is the array that holds SELECT sub queries */
11:     selectQueries[i] ← createSelectQ(triple)
12: end for
13:
14: /* Fill the query-IP Matrix */
15: for all askq in askQueries[] do
16:     for all ip in IPList[] do
17:         /* runAtRemoteIP is the method that sends the query to remote end which simply runs it on SPARQL Endpoint */
18:         /* qIPMatrix[][] is the matrix that is holding the IP's and whether they can return a result for a sub Query "askq". */
19:         qIPMatrix[IP][askq] ← runAtRemoteIP(ip, askq)
20:     end for
21: end for
22:
23: /* Calculate query cost for each SELECT sub query on Hadoop and insert queries into a list which was sorted wrt. cost */
24: for all selectq in selectQueries[] do
25:     /* sortedSelectQ[] is a list which holds the SELECT queries sorted wrt. their cost values. First element is the cheapest. */
26:     /* costOfQuery is a hashmap holding the cost value for each SELECT query. */
27:     costOfQuery[selectq] ← calculateCost(selectq)
28:     sortedSelectQ[] ← createSortedList(costOfQuery[selectq])
29: end for
```

```
30: /* Prepare the query-IP stack which will be delegated to related cluster nodes */
31: /* queryIPStack[] is the stack object which will hold (query,IP) binaries. */
32: queryIPStack[] ← newStack()
33: for i = 0 to sortedSelectQ.size() do
34:     query ← sortedSelectQ[i]
35:     runnableIPs[] ← getIP(qIPMatrix, query)
36:     for all ip in runnableIPs[] do
37:         queryIPStack.add(query, ip)
38:     end for
39: end for
40:
41: /* Determine IPs of the cluster nodes to delegate the modified Query-IP Stack */
42: /* Get the peek element of the stack */
43: queryIPElement ← queryIPStack.peek()
44: /* Get the query component of the peek element */
45: peekQuery ← queryIPElement.getQuery()
46: /* Get(by removing) all of the element which includes peekQuery from queryIPStack */
47: topLevelElements[] ← getElementsWithQuery(queryStack, peekQuery)
48: for all element in topLevelElements[] do
49:     delegationIPs[] ← element.getIP()
50:     /* Create modified stacks */
51:     /* modifiedQIPStack[] is the stack which was modified for the delegation IP */
52:     modifiedQIPStack[] ← createModifiedQIPStack(queryIPStack, element.getQuery(), element.getIP())
53:     /* Create a hashmap holding delegation IP and its modified stack */
54:     /* modifiedQIPStackMap is a hashmap holding the modified stack for each delegation IP. */
55:     modifiedQIPStackMap[] ← newHashMap()
56:     modifiedQIPStackMap.put(ip, modifiedQIPStack[])
57: end for
58:
59: /* Delegate Query Stack to Cluster-Nodes and get results */
60: for all ip in delegationIPs[] do
61:     Results[ip] ← delegateTo(ip, queryIPStack)
62: end for
```

**Algoritma 2** DRS Uç Birim Uygulamalarının SPARQL Sorgu Yığıtlarının Çalıştırması

**Input:** A Query-IP stack: *QIPStack*
**Output:** A joined answer for Query-IP stack QIPStack: *StackResult*

```
1:
2:  /* Get query for this cluster node from Query-IP Stack */
3:  for all queryIpElement in QIPStack do
4:      if this.ip == queryIpElement.getIP() then
5:          /* localQuery is the query extracted from QIPStack that is targeting this cluster node. */
6:          localQuery ← queryIpElement.getQuery())
7:          QIPStack.remove(queryIpElement)
8:      end if
9:  end for
10:
11: /* Determine the IP's of the cluster nodes to delegate the modified Query-IP Stacks */
12: /* Get the peek element of the stack */
13: queryIPElement ← QIPStack.peek()
14: /* Get the query component of the peek element */
15: peekQuery ← queryIPElement.getQuery()
16: /* Get (by removing from the stack) all of the element which includes peekQuery */
17: topLevelElements[] ← getElementsWithQuery(QIPStack, peekQuery)
18: for all element in topLevelElements[] do
19:     delegationIPs[i] ← element.getIP()
20:     /* Create modified stacks */
21:     /* modifiedQIPStack[] is the stack which was modified for the delegation IP */
22:     modifiedQIPStack[] ← createModifiedQIPStack(queryIPStack, element.getQuery(), element.getIP())
23:     /* Create a hashmap holding delegation IP and its modified stack */
24:     /* modifiedQIPStackMap is a hashmap holding the modified stack for each delegation IP. */
25:     modifiedQIPStackMap[] ← newHashMap()
26:     modifiedQIPStackMap.put(ip, modifiedQIPStack[])
27: end for
28:
29: /* Delegate Query-IP Stacks to related cluster nodes. */
30: for all ip in delegationIPs do
31:     /* remoteAnswers is the query answer array to the delegated query-IP stack. */
32:     remoteAnswers[ip] ← delegateTo(ip, modifiedQIPStackMap(ip))
33: end for
34:
35: /* Run local query by sending it to the local Fuseki SPARQL end point*/
36: /* localAnswer is the answer returned from the local Fuseki SPARQL end point. */
37: localAnswer ← runLocalQuery(localQuery)
38:
39: /* Join local and remote answers */
40: StackResult ← joinAnswers(localAnswer, remoteAnswers[])
```

#### 3.2.3.6 Alt SPARQL Sorgu Sonuçlarının Birleştirilmesi

Dağıtık bir sistemin üçlü veritabanı sisteminin sağlaması gereken en önemli özellik, kullanıcıya geri gönderilecek olan cevapların, tek başına çalışan bir SPARQL sorgu uç biriminden dönecek olan cevaplar ile aynı olmasıdır. Ancak, dağıtık olarak depolanan veri parçalarının, dağıtık olarak sorgulanması durumunda, sorgunun, hangi alt SELECT sorgusu olduğuna ve hangi küme uç birimi üzerinde yürütüldüğüne bağlı olarak, farklı yapıda ve farklı içerikli cevaplar oluşabilmektedir. Dolayısı ile, farklı uç birimlerden elde edilen alt sorgu cevaplarının birleştirilmesi çok büyük önem taşımaktadır.

Bu bölümde, Şekil 3.15'de gösterilen cevaplama sürecinde, cevapların uç birimlerden kök birime doğru ilerlerken nasıl birleştirildiği açıklanacaktır.

Tasarlanan sistemde uç birimlere gönderilen sorgular SELECT sorgularıdır. Fuseki SPARQL uç birimi, SELECT sorgularına, birer tablo şeklinde cevap gönderebilmektedir. Örneğin, SPARQL 3.1 sorgusunun, tüm verilerin birlikte depolandığı bir üçlü veritabanında, yürütülmesi durumunda, Fuseki SPARQL uç biriminden geri gönderilen cevap, mantıksal seviyede Çizelge 3.4'de verildiği gibi bir tablodur. Bu sorgu örneğinde, iki farklı değişken sorgulanmıştır. Değişken isimleri "?X" ve "?Y" olan bu değerler sırası ile, öğrencilere ve derslere karşılık gelmektedir. SPARQL 3.1 sorgusu, bu iki değişkeni birbirine bağlayan bir üçlü içerdiği için, cevap içinde dönen "X" ve "Y" değerlerinin sayıları aynıdır.

Çizelge 3.4: Örnek SPARQL SELECT sorgusu cevabı

| X | Y |
|---|---|
| Student11 | calculus101 |
| Student23 | calculus102 |
| Student43 | Differential Equations |
| ... | ... |
| Student56 | Quantum Mechanics |

Bu tez çalışmasında önerilen yaklaşım, orijinal sorgunun üçlü bileşenlerine ayrılması ve bunların kullanılması yoluyla yeni sorgular üretilmesi üzerine kurulu

olduğu için, bir SPARQL uç birimine gönderilen sorgular, her zaman tek üçlü içerecektir. Doğal olarak farklı küme uç birimleri, aynı sorguya farklı sayıda değer içeren cevaplar üretebilirler. Cevap olarak alınan tabloda kaç sütün bulunacağı ilgili sorguda kaç adet değişken sorgulandığına göre değişebilmektedir. Dolayısı ile, bir sorgunun sağlaması gereken üçlülerden oluşturulan alt sorgular, farklı uç birimlerden, Çizelge 3.4'de gösterilen yapıda, ancak sütün sayısı ve içeriği değişebilen cevaplar almaktadırlar.

Birleştirme işleminin, DRS uç birim uygulamalarında nasıl yürütüldüğü, aşağıda maddeler halinde verilmiştir. Açıklanan süreç, bir DRS uç birimine ait sorgu sonuçları birleştirme biriminin, yerel sorgu cevabını ve delege ettiği yığıta[13] verilen cevabın birleştirilmesi sürecini anlatmaktadır.

1. Birleştirme işlemi sonucunda, oluşturulacak olan cevap içinde bulunacak değişkenlerin belirlenmesi

   (a) DRS uç birim uygulaması, sorgu yönetim bileşeni tarafından, yerel Fuseki SPARQL uç birimi üzerinde yürütülen sorguya verilen cevap içinde bulunan değişken isimleri okunur ve birleştirme sonucu oluşturulacak cevabın değişken isimleri listesine eklenir. Yerel Fuseki SPARQL uç birimine gönderilen her sorgu içinde, sadece bir adet üçlü bulunacağı için, bu uç birimden dönen cevap içinde en az 1, en fazla 2 sütun olabilir. Örnek olarak, SPARQL 3.1 sorgusundan üretilen alt sorguların, yerel Fuseki SPARQL uç birimi tarafından cevaplanması durumunda, cevabın içerebileceği değişken isimleri Çizelge-3.5 içinde verilmiştir. Bu alt sorgulardan hangisi cevaplanmış ise ona ait değişken değerleri, birleştirme süreci sonunda kullanılacak olan değişken listesine eklenir.

Çizelge 3.5: SPARQL 3.1 sorgusuna ait alt sorguların değişkenleri

| Alt Sorgu | Değişken İsimleri |
|---|---|
| S1 : ?X rdf:type ub:GraduateStudent | X |
| S2 : ?X ub:takesCourse ?Y | X,Y |
| S3 : ?Y rdf:type <http://www.Department0.University0.edu/GraduateCourse0> | Y |

[13]Yığıta cevap veren küme uç birimleri de, burada anlatılan adımları işletmişlerdir.

(b) Yerel sorgudan elde edilen değişken isimleri, birleştirme sonucu oluşacak cevabın değişken listesine eklendikten sonra, gönderilmiş olan sorgu yığıtına gelen cevap içindeki değişken isimleri de aynı listeye eklenmelidir. Birleştirme işlemine, kaç adet yığıt cevabının gireceği, birleştirme yapan küme uç biriminin, kaç farklı küme uç birimine sorgu yığıtı gönderdiğine bağlıdır. Örneğin, Şekil-3.14 içinde gösterilen, IP2 küme uç birimi, IP1, IP3 ve IP5 küme uç birimlerine farklı yığıtlar gönderilmiştir. Dolayısı ile IP2 üzerinde yapılacak olan birleştirme işlemine 3 adet yığıt cevabı katılacaktır. Yığıt cevapları, Fuseki SPARQL uç birimlerinin içerdiği gibi en az 1, en fazla 2 sütun içermek zorunda değillerdir. Çünkü yığıt cevaplarının her biri, sorgu dağıtım ağacında daha alt seviyelerden gelen farklı sorgu cevaplarının birleştirilmesi biçiminde oluşturulmuştur. Ancak SPARQL 3.1 sorgusu için bu değerler, Fuseki SPARQL uç birimindekiler ile aynıdır. Eğer örnek olarak SPARQL 3.1 yerine, Ek-B içinde verilen SPARQL-B.2 sorgusu kullanılsa idi, birleştirmeyi yapan DRS uç birimine ait sorgu birleştirme bileşenine, yerel Fuseki SPARQL uç biriminden gelecek olan cevabın değişkenleri {X, Y, Z, XY, YZ, XZ}[14] kümesinin elemanlarından birisi olacak iken, yığıtlardan gelen cevaplar, {X, Y, Z, XY, YZ, XZ, XYZ} kümesinin elemanlarından birisi olacaktır. Dolayısı ile bu adımda, DRS uç birimine ait sorgu birleştirme bileşeni, yığıt cevapları içindeki değişken isimlerini tespit ederek, bunları birleştirme sonucu oluşacak olan cevabın değişken listesine ekler.

2. Madde-1 içinde belirlenen değişkenlere göre birleştirme işleminin yapılması

   (a) Kullanılacak olan değişken isimleri, farklı noktalardan gelen cevaplar içinde bulunan kümelerin kesişimlerinin bulunması amacı ile kullanılacak olan listedir. İlk olarak, DRS uç birimine ait sorgu birleştirme bileşeni bu listedeki değişken isimlerinden ilkini okur. Sonraki adımlar, bu madde içinde ele alınacak olan diğer liste elemanları üzerinden bir döngü içinde ele alınacaklardır.

[14]Burada gösterilen kümenin, iki ve daha fazla değişken içeren elemanlarının, değişkenlerinin sıralaması önemli değildir. XY ve YX aynıdır.

(b) Okunan değişken ismini içeren cevap kümeleri tespit edilir. Örneğin, okunan değişken ismi X ise ve birleştirme işlemine girecek yerel sorgu cevabı ve yığıt cevapları sırasıyla X, XY ve Y kümeleri ise, bu adımda kullanılacak olan kümeler X ve XY kümeleridir. Çünkü sadece bu iki cevap kümesi X değişkenini içermektedir.

(c) Tespit edilen X ve XY kümeleri, Madde-2a adımında tespit edilen edilen sütün üzerinden kesiştirilir. Kesişim işlemi X kümesinde bulunan tüm elemanların, XY kümesinin, X sütununda bulunup bulunmadığına göre yapılır. Kesişim bulunması durumda, kesişen X değerine karşılık gelen Y değeri de okunur[15] ve kesişim değerlerini tutmak amacı ile kullanılan ara veri yapısı içine kaydedilir. Şekil-3.16 bu adımda gerçekleştirilen işlemi göstermektedir.

| X | Y |
|---|---|
| Student11 | Course1 |
| Student12 | Course2 |
| Student13 | Course3 |
| Student14 | Course4 |
| Student15 | Course5 |
| Student16 | Course6 |
| Student17 | Course7 |

(a) Örnek XY kümesi

| X |
|---|
| Student11 |
| Student12 |
| Student13 |
| Student14 |

(b) Örnek X kümesi

| X | Y |
|---|---|
| Student11 | Course1 |
| Student12 | Course2 |
| Student13 | Course3 |
| Student14 | Course4 |

(c) Birleştirilmiş küme

Şekil 3.16: İki kümenin kesiştirilmesi - 1

(d) Madde-2b içinde kullanılan kümeler, belli bir değişken üzerinden kesiştirildiklerinden dolayı, döngünün devamında, kesişime giren kümeler yerine birleşim kümesi kullanılır. Bu sayede farklı değişkenler göz önüne alınarak kesişimler oluşturulurken, aynı değerlerin, ilk döngü turunda kesişim dışında kalmış olmalarına rağmen tekrar ele alınmalarına engel olunmuş olunur.

[15]Burada Y değerininde alınmasının sebebi, XY kümesini cevap olarak gönderen küme uç birimine gönderilen sorguyu, X ve Y'nin birlikte aynı anda sağlamış olmalarıdır.

(e) İlk değişken için kesişim kümesi bulunduktan sonra, sonraki değişken ile aynı işlemlerin yapılması için Madde-2a'e dönülür işlem adımları aynı sırada tekrarlanır. Örnek olarak, Madde-2a'de ikinci değişken olarak olarak Y'nin okunması durumunda, Madde-2b'de Y ve XY kümeleri tespit edilecektir. Ancak Madde-2d içinde anlatıldığı gibi XY kümesi, X için yürütülen ilk döngü turunda kullanıldığı için, o küme yerine X ve XY kümelerinin kesişimi kullanılır. Benzer şekilde, Y kümesi ve X-XY kesişimi Y sütunu üzerinde ortak elemanları bulmak için kesiştirilir. X-XY kesişimi bulunurken yapıldığı gibi, Y ve x-XY kesişiminin Y sütununda ortak eleman bulunması durumunda, ona bağlı olan X değeri de alınır. Şekil-3.17 bu adımda gerçekleştirilen işlemi göstermektedir.

| X | Y |
|---|---|
| Student11 | Course1 |
| Student12 | Course2 |
| Student13 | Course3 |
| Student14 | Course4 |

(a) Birinci döngü turunda bulunan, X ve XY kümelerinin kesişimi

| Y |
|---|
| Course2 |
| Course3 |

(b) Örnek Y kümesi

| X | Y |
|---|---|
| Student12 | Course2 |
| Student13 | Course3 |

(c) Birleştirilmiş küme

Şekil 3.17: İki kümenin kesiştirilmesi - 2

(f) Birleştirilmeleri için, DRS uç birim uygulaması sorgu birleştirme bileşenine gelen cevapların ayrık kümeler olması durumunda işlemde hiç bir fark olmayacaktır. Örneğin birleştirme işlemine giren kümeleri X, Y1 ve Y2[16] şeklinde olmaları durumunda, ilk adımda, cevapta olması gereken değişken isimleri belirlendiğinde yine X ve Y değerleri elde edilecektir. Ancak birleştirme safhasında, X değişkeni ele alındığında, kesişimi alınabileceği, X değerleri içeren başka bir küme olmadığı için, X kümesinin kendisi birleşim kümesi gibi alınarak,

[16]Burada değişken isimler Y'dir. Y harfinin yanında görülen 1 ve 2, sadece ayrı iki Y kümesi geldiğini göstermek amacı ile kullanılmıştır.

oluşturulacak cevaba yerleştirilir. Ancak, Y kümeleri kendi aralarında kesiştirilebileceklerinden dolayı, oluşturulacak olan cevaba kesişimleri eklenecektir. Dolayısı ile yukarıda anlatılan duruma göre tek fark, gönderilen cevabın, ayrık iki küme içeriyor olmasıdır. Birleştirme işleminin yapıldığı küme uç biriminde ayrık olan kümeler, cevabın gönderildiği bir üstte birleştirme için kullanılacak olan diğer kümeler aracılığı ile ilişkili biçime dönüşebilirler. Ancak bu bir zorunluluk değildir. Orijinal sorgu, ayrık iki değişkeni içeren bir sorgu da göndermiş olabilir. Burada bahsedilen, her iki türlü cevabı da, aynı anda göndermeyi sağlayacak olan bir veri yapısı uygulama sırasında tasarlanmıştır. Şekil-3.18 bu adımda gerçekleştirilen işlemi göstermektedir.

| X |
|---|
| Student11 |
| Student12 |
| Student13 |
| Student14 |

(a) Örnek X kümesi

| Y |
|---|
| Course2 |
| Course3 |

(b) Örnek Y kümesi

| X | Y |
|---|---|
| Student11 | Course2 |
| Student12 | Course3 |
| Student13 | |
| Student14 | |

(c) Birleşime giren kümelerin ayrık olması durumunda cevap olarak gönderilecek küme

Şekil 3.18: Ayrık kümelerin birleştirilmesi

(g) Birleştirme işlemi sonucunda elde edilen küme veya kümeler, Şekil-3.15 içinde verildiği gibi bir üst uç birime gönderilir. Benzer döngüler, sonuçlar DRS merkez sunucusuna ulaşana kadar devam eder. DRS merkez sunucusu sonuçları, ölçülen zaman değerleri ile birlikte, kullanıcıya gösterilmek üzere komut satırı bileşenine gönderir.

# 4. PERFORMANS DEĞERLENDİRME

Bu bölümde, ilk olarak, sistem üzerinde yürütülen deneylerde kullanılan veri hakkında bilgi verilecek, daha sonra deneylerin ve alınan sonuçların detayları açıklanacaktır.

## 4.1 Kullanılan Veri ve Yapılan Deneyler

Deneyler sırasında, LUBM (Lehigh University Benchmark) veri üreticisi tarafından üretilen veri kümeleri kullanılmıştır [46]. LUBM, semantik ağ çalışmaları sırasında ihtiyaç duyulan büyük veri kümelerinin, yapay olarak oluşturulmasını sağlayan bir yazılımdan, bu yazılım ile birlikte kullanılabilecek bir ontolojiden ve test amacı ile kullanılabilecek bir sorgu kümesinden oluşmaktadır. Kullanılan bu sorgular ve ontoloji, sırası ile Ek-B ve Ek-C içinde verilmiştir. Deney amacı ile veri üretim işleminin detayları ise Ek-A bölümünde açıklanmıştır. LUBM test araçlarının geliştirilmesinde en önemli gerekçelerden bir tanesi, farklı çalışma gruplarının, çalışmalarını ortak bir referans etrafında geliştirebilmesini sağlamaktır [21]. Ancak, sonuçların doğrudan karşılaştırılması çok doğru değildir. Kullanılan donanımsal altyapı, ağ topolojisi gibi parametreler benzer uygulamaların, çok farklı sonuçlar üretmesine neden olabilmektedir. Dolayısı ile doğru bir karşılaştırma, ancak bazı parametrelerin sabit tutulduğu kontrollü bir deney ortamında yapılabilir.

LUBM, veri üretimi için, bir üniversite ve bileşenlerinin bulunduğu, ve LUBM'un parçası olan, "univ-bench.owl" isimli ontoloji dosyasını kullanır. Üretilecek olan veri büyüklüğü, amaca göre, kullanıcı tarafından bir girdi olarak verilebilmektedir. Üretilen RDF verisi OWL yapısında olmaktadır. Tez çalışması sırasında geliştirmiş olan sistem, N-Triple yapısını kullandığı için, üretilen veri N-Triple yapısına çevrilmiştir. Bu çevirme işleminin detayları da Ek-A bölümünde bulunmaktadır.

Yapılan deneylerde, sistemin başarısı değerlendirilirken, genel olarak iki nokta göz önüne alınmıştır. Birincisi, sistemin sorguları doğru cevaplayabilme yeteneği, ikincisi ise verilen cevapların, kullanıcıya dönme sürelerinin veri büyüklüğü ve küme büyüklüğüne olan bağlılıklarının ölçülmesidir.

### 4.1.1 Cevap Doğruluk Deneyleri

Yukarıda bahsedilen birinci deney, yani sistemin sorguları doğru cevaplayabilme yeteneğinin ölçülmesi, LUBM test sorgularının kullanılması yolu ile denenmiştir. LUBM, Ek-B içinde detayları verilmiş olan, 14 adet SPARQL sorgusu içermektedir. Bu sorgular LUBM ile dağıtılan ontolojiden üretilen veriler göz önüne alınarak oluşturulmuştur. Sorguların her birisi, test edilen üçlü veritabanının farklı soru biçimlerine verdiği tepkiyi görebilmek amacı ile tasarlanmışlardır. Yapılan her bir sorguya karşı, alınan cevapların doğru olup olmadığının tespit edilmesi amacı ile, dağıtık olmayan bir sistemin referans alınması gerekmektedir. Bu amaçla, üretilen tüm veriyi tek başına depolayan, ayrı bir sorgu uç birimi hazırlanmıştır. Dağıtık sistem üzerinde çalıştırılan her sorgunun sonucu, aynı sorgu için referans sunucudan dönen cevaplar ile karşılaştırılmıştır. Bu deneylerde performans farkları göz ardı edilmiş, sadece sistemin doğru cevap verebilme yeteneği test edilmiştir.

Yapılan bu deney sonucunda alınan cevaplar, referans sistemden dönen cevaplar ile birebir örtüşmektedir. Bu sonuç, tasarımın cevapları oluşturma aşamasında doğru çalıştığını göstermektedir.

LUBM tarafından sağlanan sorguların 3 tanesi (sorgu-11, sorgu-12, sorgu-13), sistemin çıkarım yapabilme (Inference) yeteneğini ölçmek amacı ile yazılmıştır. Ancak, Fuseki SPARQL uç birimlerinde çıkarım yapabilme yeteneğinin kullanılabilmesi için gereken donanımsal gereksinimler, bu tez çalışmasında kullanılan deney ortamı tarafından sağlanamadığı için, bu sorgular denenmemiştir. Çıkarım işlemi yoğun bellek ve işlemci kullanımı gerektiren bir işlemdir.

LUBM test sorguları dışında, yapılan sorgularda genel olarak başarı elde edilmektedir. Ancak, geliştirmiş olduğumuz DRS uygulaması, W3C tarafından, SPARQL için hazırlanan öneri dokümanının tamamını uygulamadığından, başarısız olunan sorgular bulunmaktadır. Ancak bunlar, sistemin tasarımı ile ilgili hatalardan dolayı değil, başlangıçta, uygulanması hedeflenmeyen SPARQL yetenekleri gerektirdikleri için cevaplanamayan sorgulardır. Tez çalışmasının ilk hedefi, temel sorguları dağıtık ortamda cevaplamak olduğu için, deneyler sadece LUBM test sorguları ile sınırlı tutulmuştur.

### 4.1.2 Cevap Dönüş Süreleri Karşılaştırma Deneyleri

Dönüş sürelerinin, dağıtık olmayan bir sisteme göre kazancının ne seviyede olduğunun ve bu değerlerin veri ve küme boyutuna nasıl bağlı olduğunun ölçülmesi amacı ile, dağıtık sisteme gönderilen her sorgu, referans sunucuya da gönderilmiştir. İki sistemde de sorguların başlamasından cevapların alınmasına kadar geçen zamanlar ölçülmüştür. Elde edilen sonuçlar, tablolar halinde gösterilmiştir. Ayrıca, dağıtık sistemdeki, küme bilgisayar sayısının ve veri büyüklüğünün etkileri ayrı ayrı denenmiş ve değişimler tablolarda gösterilmiştir.

Küme bilgisayar sayısının etkisini tespit etmek amacı ile, sırası ile 5, 10 ve 15 uç birimin bulunduğu küme konfigürasyonlarında sorgular çalıştırılmıştır. Elde edilen sonuçlar referans sunucu için elde edilen değerleri de içerecek şekilde, karşılaştırmalı olarak verilmiştir.

Veri büyüklüğünün etkisini gözlemlemek amacı ile 2GB, 4GB, 6GB, 8GB ve 10GB'lık deney verileri ile test yapılmıştır. Semantik Web alanında yapılan

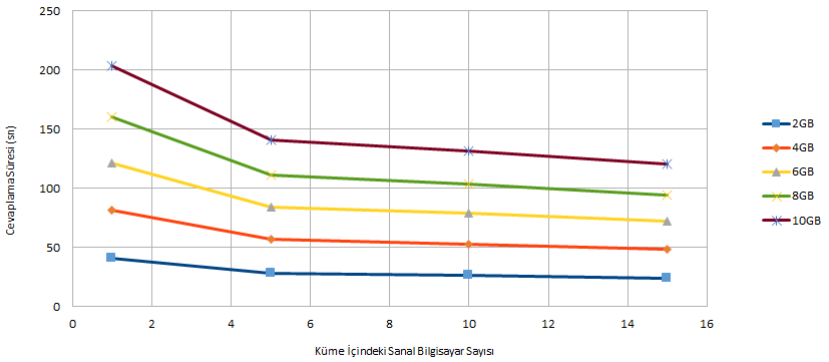
çalışmalar için küçük sayılacak olan bu veri büyüklükleri, kullanılan sistemlerin kısıtları dolayısı ile bu değerlerde seçilmişlerdir.

Deneylerin kontrollü olarak yapılması amacı ile, küme bilgisayar sayısının cevap verme süresine etkisinin ölçülmesi sırasında, veri büyüklüğü sabit tutulurken, veri büyüklüğünün etkisi ölçülürken küme içindeki bilgisayar sayısı sabit tutulmuştur. Aşağıdaki bölümlerde her deneyin ayrı ayrı çıktıları gösterilmiştir.

#### 4.1.2.1 Cevaplama Sürelerinin Veri Büyüklüğü İle İlişkisi

Veri büyüklüğünün artması, sorgu cevaplama süresinin artmasına sebep olmaktadır. LUBM sorgularından birincisi(Ek-B, LUBM Test Sorgusu 1) kullanılarak, farklı veri büyüklüklerinde yapılan cevaplama süresi testlerinin sonucu Şekil 4.1'de verilmiştir. Bu şekilde her renk farklı boyuttaki veri kümesi üzerinde yapılan sorgu sürelerini göstermektedir. Sonuçlar yatay eksende, farklı büyüklükteki kümelerde denenmiş olarak gösterilmektedir. Şekilde küme birim sayısının arttırılmasının, sorgu cevaplama sürelerine olan katkısının, birinci değer dışında çok büyük bir fark yaratmadığı görülmektedir. Bu durumun, farklı sorgular ile yapılan deneylerde de benzer şekilde sonuçlar ürettiği, ancak süre değerlerinin farklı olduğu gözlemlenmiştir. Bunun sebebinin, fiziksel sunucular ile kurulan bir küme yerine, sanallaştırılmış sunucular kullanılması olduğu düşünülmektedir. Sanallaştırılmış sunucular, her ne kadar kendi kullanımları için ayrılmış olan işlemci çekirdeklerine ve belleğe sahip olsalar da, sanallaştırmayı yöneten işletim sistemi açısından, tüm sanal makinalar yürütülen ayrı bir işlemdir. Sanallaştırmayı yöneten işletim sistemi, işlemci kullanım hakkını, sanal makinalara başka hiç bir işlemin çalıştırılamayacağı şekilde tahsis etmemektedir. Dolayısı ile sanallaştırmayı yapan işletim sistemi, tüm sanal sunucuların aynı anda işlemciyi yoğun bir şekilde kullanmaya başlaması durumunda, çok yoğun bir şekilde, bağlam anahtarlama (context switch) yapmaktadır. Bu önemli bir zaman kaybı oluşturmaktadır. Deneylerin gerçek fiziksel sunucularda çalıştırılması durumunda, bu tür bir zaman kaybı olmayacaktır. Dolayısı ile küme içindeki bilgisayar sayısının arttırılmasının, cevaplama sürelerine katkısının daha yüksek

olması beklenebilir.



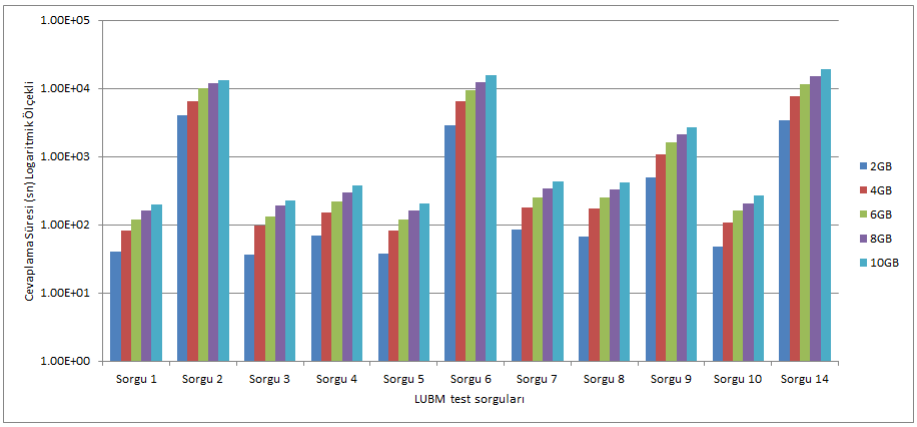

Şekil 4.1: LUBM Sorgu1 için farklı veri büyüklerinde ve farklı küme birim sayısında cevap süreleri.

Şekil 4.1'e karşılık gelen veri sonuçları Çizelge 4.1'de verilmiştir.

Çizelge 4.1: LUBM Sorgu1(Ek-B) için farklı küme birim sayıları ve farklı veri büyüklüklerinde sorgu cevaplama süreleri (sn).

| Veri Büyüklüğü/ Birim Sayısı | 1 | 5 | 10 | 15 |
|---|---|---|---|---|
| 2GB | 40.31 | 28.43 | 25.58 | 23.25 |
| 4GB | 83.89 | 57.79 | 56.09 | 47.45 |
| 6GB | 121.43 | 83.21 | 79.72 | 73.84 |
| 8GB | 160.07 | 114.00 | 102.77 | 92.70 |
| 10GMB | 201.12 | 144.16 | 135.97 | 123.43 |

Tüm sorguların, farklı veri boyutları ile, tek sunucuda sorgulanması sonucu elde edilen cevap süreleri ise Şekil 4.2 ve Çizelge-4.2 içinde verilmiştir. Sonuçların dikey eksende logaritmik (logarithmic growth) şekilde verilmesinin sebebi, bazı sorgu sonuçlarının çok uzun sürmesi dolayısı ile çok büyük değerlere sahip olması ve diğer değerler ile karşılaştırmalı olarak gösteriminin anlaşılır olmamasıdır. Sorgu 11, 12 ve 13 , üçlü veritabanının çıkarım yapmasını gerektirdiği ve deney donanımının bu işlemi desteklemiyor olması dolayısı ile verilen şekilde görünmemektedirler.

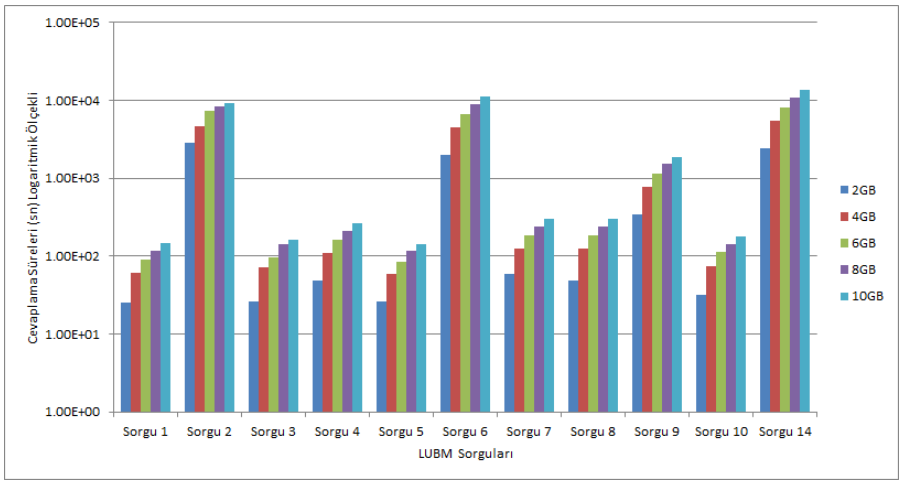



Şekil 4.2: Tüm sorgu sonuçlarının tek sunucu üzerinde karşılaştırması

Çizelge 4.2: LUBM sorgularının tek bilgisayar üzerinde, farklı veri büyüklüklerinde, cevaplanma süreleri (sn).

| Veri | Sorgular | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 14 |
| 2GB | 40.31 | 4071.05 | 36.74 | 69.62 | 37.83 | 2893.20 | 85.66 | 67.91 | 495.77 | 48.29 | 3471.42 |
| 4GB | 83.89 | 6475.73 | 97.77 | 153.42 | 83.27 | 6415.20 | 177.25 | 171.52 | 1095.35 | 109.12 | 7697.43 |
| 6GB | 121.43 | 10160.16 | 133.01 | 223.45 | 120.23 | 9514.49 | 256.35 | 249.68 | 1610.12 | 165.12 | 11416.16 |
| 8GB | 160.07 | 11903.11 | 194.23 | 299.15 | 163.52 | 12543.15 | 341.56 | 332.63 | 2124.96 | 205.25 | 15044.61 |
| 10GB | 201.12 | 13062.19 | 229.97 | 376.65 | 206.08 | 15945.42 | 432.88 | 421.80 | 2685.85 | 267.35 | 19132.03 |

Şekil 4.3'de, aynı sorguların 5 birimli küme ile sorgulanması sonucu elde edilen sonuçlar bulunmaktadır. Sonuçlar Şekil 4.2'da elde edilen sonuçlarla örtüşmektedir. İki şekil arasındaki en büyük fark sürelerin azalmış olması ancak, sorguların cevap sürelerinin orantısal olarak yakın değerlere sahip olmasıdır. Dolayısı ile veri büyüklüğünün arttırılmasının, yapılan sorgudan bağımsız olarak sorgu sürelerini uzatmış olduğu, tek sunucu ve 5 birimli bir kümede gösterilmiş olmaktadır. Ayrıca, aynı şekil ve çizelge, veri boyutunun sabit tutulduğu durumlarda, küme kullanımının zamansal bir avantaj sağladığı da görülmektedir. Şekil 4.3'e ait deney verileri Çizelge 4.3'de verilmiştir.




Şekil 4.3: LUBM sorgularının 5 birimli küme üzerinde, farklı veri büyüklüklerinde, cevaplanma süreleri (sn).

Çizelge 4.3: LUBM sorgularının 5 birimli küme üzerinde, farklı veri büyüklüklerinde, cevaplanma süreleri (sn).

| Veri | Sorgular | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 14 |
| 2GB | 25.22 | 2817.49 | 25.96 | 48.27 | 26.15 | 1981.75 | 59.25 | 47.93 | 343.03 | 32.41 | 2401.60 |
| 4GB | 61.77 | 4618.95 | 71.21 | 109.62 | 59.33 | 4528.77 | 126.35 | 124.76 | 781.11 | 75.48 | 5488.31 |
| 6GB | 89.86 | 7283.63 | 97.37 | 160.47 | 86.10 | 6750.69 | 183.67 | 182.53 | 1154.01 | 114.79 | 8180.98 |
| 8GB | 117.68 | 8477.49 | 141.26 | 213.43 | 116.34 | 8841.56 | 243.12 | 241.58 | 1513.08 | 141.76 | 10710.90 |
| 10GB | 145.86 | 9177.05 | 164.99 | 265.09 | 144.64 | 11087.62 | 303.95 | 302.20 | 1886.58 | 182.15 | 13436.50 |

#### 4.1.2.2 Cevaplama Sürelerinin Küme Büyüklüğü İle İlişkisi

Kümede bulunan, uç birim sayısının arttırılması sonucu, birim sunucu başına düşen veri miktarı azalmaktadır. Bu sayede, her uç birim için, veri arama süresi azalmaktadır. Ancak, verinin daha çok parçalanması sebebi ile, ağ üzerinde yapılacak olan haberleşme miktarı artmaktadır. Haberleşme miktarının (paket gönderme alma sayısı) artıyor olmasına karşın, gönderilip alınan paket büyükleri azalmakta ve haberleşen uç birim sayısı arttığı için bu paketlerin işlenmesi yükü, her birim üzerinde daha az olmaktadır.

Bu deneyde kümedeki birim sayısının sorgu süreleri üzerindeki etkisi incelenmiştir. Deney, ortaya koyduğu sonuçlar itibari ile, ağ kullanım yoğunluğunun

sebep olduğu zaman artışlarının, verinin küçülmesi ile oluşan zaman azalmalarına oranla etkisinin ne yönde olduğunu göstermektedir. Eğer verinin küçülmesinin getirdiği artı etkiler, eksi etkilere oranla daha fazla ise toplam sürede bir azalış, tersi durumda ise yükseliş görüleceği öngörülmüştür. Şekil 4.1'de LUBM Sorgu-1 için verilen sonuçlar, verinin küçülmesinin getirdiği zaman kazançlarının 5, 10 ve 15 birimli küme için daha az olduğu görülmektedir. Bu sonuca dayanarak, veri boyutu küçüldükçe, işleme amacı ile küme kullanımının getirisinin azaldığı söylenebilir.

Şekil 3.14'de verilen sorgu yığıtı dağılım yapısında bulunan her hangi bir ara birimin, kendi çocuklarından cevap alma süresi, genel olarak $T_{Max} = \max(t_i) + \max(s_i) + Mt$ şeklinde ifade edilebilir. Bu ifadede, $T_{Max}$ bir birimin, alt birimlerden beklediği cevapları alma ve kendi yerel sorgu sonuçları ile birleştirip, kendi göndereceği cevabı hazırlama süresini gösterirken, $\max(s_i)$ bu birimin cevap beklediği alt birimlerin, veri arama-birleştirme süreleri içinde en büyük olanını, $\max(t_i)$ alt birimlerin ağ üzerinden cevap gönderme sürelerinin en büyük olanını,$Mt$ ise verilerin sorgu yapan birimde birleştirilmesi için geçen zamanı göstermektedir. En büyük değerlerin alınmasının sebebi, daha kısa sürede alınacak cevapların, tüm alt birimlerden cevap gelmediği sürece, toplam cevabın hazırlanma açısından bir öneminin bulunmamasıdır.

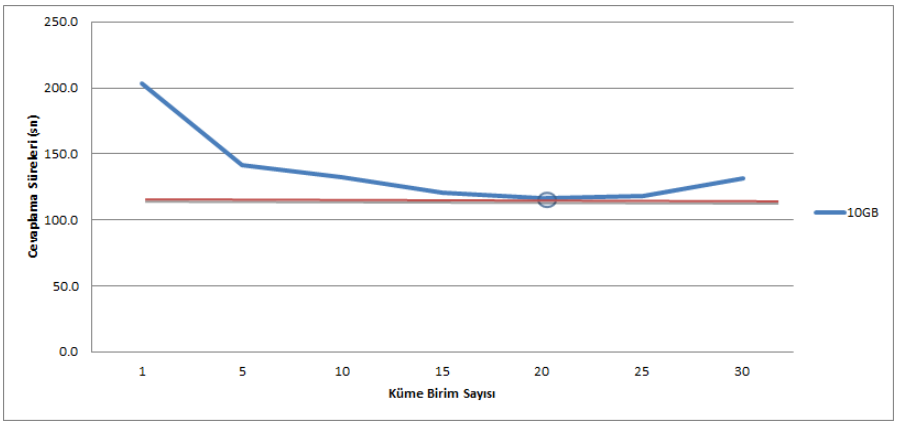
## 4.2 Değerlendirmeler

Cevap doğruluk deneyleri sonucunda beklenen cevapların alınıyor olması, tasarımın doğru çalıştığını göstermektedir. Bu ise, sorgu parçalama , alt sorguların uç birimler üzerinde çalıştırılması ve ana sorgu biriminin, uç birimlerden gelen sorgu sonuçlarını birleştirme işlemini doğru bir şekilde yaptığını ispatlamaktadır. Bu anlamda cevap doğruluk deneyleri amaçlanan hedefi sağlamıştır.

Veri büyüklüğünün sorgu cevaplama süresine etkisini ölçen deneye ait sonuçlardan, beklendiği gibi, veri büyüklüğünün artması ile sorgu cevaplama süresi artığı görülmüştür.Bu durumun sebebi, makina başına düşen veri miktarının artması

ve bu yüzden sorgu cevaplarının aranması için geçen zamanın uzamasıdır.

Cevap sürelerinin, küme büyüklüğüne bağımlılığını gösteren deney sonuçlarına göre, kümenin büyütülmesinin genel olarak bir zaman kazancı sağladığı görülmektedir. Ancak verilen Şekil 4.1'de, küme birim sayısının artması ile sağlanan kazancın giderek azaldığı görülmektedir. Buradan çıkarılabilecek olan sonuç, küme birim sayısının arttırılmasının sürekli bir kazanç sağlama yönünde olmadığıdır. Dolayısı ile her veri kümesi için, verinin parçalanması sonucu elde edilen kazançların, sistemin dağınıklığı yüzünden oluşan kayıplara eşit olduğu bir eşik birim sayısı değeri bulunabilir. Bu değerden sonra, sistemin daha fazla parçalanmasının getirdiği kazanç, kayıplardan daha fazla olacağı için, toplamda süre kazancı olmayacak ve sorgu süreleri uzamaya başlayacaktır. Deney ortamında birim sayısını daha fazla arttırma olanağı olmadığı için bu durum deneysel olarak gözlenmemiştir. Birim sayısının sürekli arttırılması durumunda cevaplama sürelerinin beklenen davranışı Şekil 4.4'de gösterilmiştir.




Şekil 4.4: LUBM Sorgu1 için cevap sürelerinin küme birim sayısına etkisi

Sorguların farklı cevaplanma sürelerine sahip olmaları, yapıları ile alakalıdır. Bazı sorgular, çizge üzerinde sadece bir sınıf değerleri üzerinden arama yaptığı halde, benzer şekilde tek sınıf üzerinden arama yapan sorulardan daha uzun zaman almaktadırlar. Bunun sebebi sorgunun, sağlaması gereken üçlü yollarının yada ilişkilerin daha fazla olmasıdır. Örneğin, Ek-B içinde verilen Sorgu-B.1 ve Sorgu-B.10, sadece "?X" sınıfı değerler için arama yapıyor olsalar da, Sorgu-B.10 içinde

sağlanmaya çalışılan ilişkilerin sayısı daha fazladır. Bu durum, sorguyu sağlayan değerlerin çizge üzerinde aranması sırasında fazladan bir işlem daha yapılması, dolayısı ile toplamda sorgunun daha uzun sürmesi anlamına gelmektedir. Bu durumun, Şekil-4.2 içinde verilen zaman değerlerine bakıldığında, Sorgu-B.10 için geçerli olduğu görülebilir.

Sorgu sürelerini etkileyen diğer bir durum, sorgunun birden fazla sınıf değeri üzerinden arama yaptığı durumdur. Birden fazla değişken sınıfı bulunan sorgular, genel olarak bu değişkenlerin birlikte sağlanması gereken üçlü yollarını da içermektedirler. Bu durum ise, verilerin birleştirilmesi gerektirdiği için, genel olarak daha fazla işlem yürütüleceği ve dolayısı ile sorgunun cevaplanması için daha fazla zaman harcanacağı anlamına gelmektedir. Sorgu-B.2 ve Sorgu-B.9 bu türdeki sorgulara güzel bir örnektir. Bu iki sorgu, üç farklı sınıf ve bunların aralarında ki ilişkiler üzerinden çalışmakta ve bu yüzden diğer sorgular nispeten daha çok zaman almaktadırlar.

# 5. SONUÇ

Tez çalışmasının temel amacı, tek sunucu üzerinde sorgulanabilen verilerin çok büyümesi durumunda, ölçeklenebilir, ancak tek sunuculu sistemlerle benzer şekilde sorgulanabilen ve performans değerleri açısından kabul edilebilir sınırlar içinde bulunan, alternatif bir dağıtık yapının önerilebilmesidir. Bu kapsamda, tek sunuculu sistemlerin beraberinde getirdiği sorunlar incelenmiş ve bunları çözmek amacı ile, dağıtık depolama ve sorgulama yöntemi önerilmiştir. Önerilen yöntem test amaçlı olarak gerçeklenmiş ve bazı deneylerle, tez hedeflerinin sağlanıp sağlanmadığı tespit edilmiştir.

Elde edilen sonuçlar doğrultusunda, önerilen yöntemin, alt birimlerden dönen cevapların doğruluğu açısından, tek sunucu üzerinde çalışan bir sistem ile aynı ve sorgu cevap süreleri açısından bir miktar daha iyi sonuçlar verdiği görülmüştür. Dağıtık bir yapıda veri depolamanın ve sorgulamanın, sorgu sürelerini azaltmasının, ancak, optimal sayıda küme birimi kullanılarak sağlanabileceği tespit edilmiştir. Ancak büyük verilerin, dağıtık ortamda daha iyi sonuçlar veriyor olmasından dolayı, bu tezde önerilen türde bir sistemin kullanımı her halükarda tercih edilebilir. Semantik ağ konusu üzerinde yapılan akademik çalışmaların sayısının giderek artıyor olması, semantik veri üretiminin ve kullanımının artması gibi nedenler dolayısı ile, verinin dağıtık ortamda etkin bir şekilde depolanabilmesi ve sorgulanabilmesi çalışmalarının da artması beklenebilir. Bu açıdan, bu tez çalışması ile önerilen sistemin semantik veriler için üçlü veritabanı oluşturma süreçlerini anlamak açısından önemli getirileri olmuştur.

Tez çalışması süresince, yapılan geliştirmeler ve araştırmalar ışığında, önerilmiş

olan sisteme bazı iyileştirmelerin yapılabileceği düşünülmektedir. Bunlar aşağıda kısaca özetlenmiştir.

Yapılması planlanan en önemli değişiklilerden bir tanesi sistemin topolojik yapısının tam olarak simetrik hale getirilmesidir. Mevcut öneride, tüm sistem, merkezi bir sunucudan verilen sorgunun diğer uç birimlere yayılması üzerine kuruludur. Ancak tüm birimlerin özdeş olduğu bir yapıda, sorgular herhangi bir birimden başlatılabilir. Bu sayede bir yük dengeleyici arkasında çalışan sistemdeki her bir birim, birer ana sorgu bilgisayarı gibi kullanılabilir. Bu durum sistemin genel tepki verme başarımını arttıracaktır. Ancak bunun başarılabilmesi için, tüm birimlerin mevcut yük durumlarını birbirlerine iletebilecekleri bir mesajlaşma altyapısı oluşturulması gerekmektedir. Bu sayede, her bir birim sorgu göndermede kullanılabildiği gibi, diğer bileşenleri alt birim olarak kullanabilme şansına sahip olacaktır.

Verinin dağıtık olarak kullanılıyor olmasının getirdiği başka bir problem, veri yedekliliğinin ve bütünlüğünün sağlanmasının daha çok kaynak ve zaman gerektirmesidir. Çizge veritabanlarında verilerin hemen hemen tamamının ilişkili olması dolayısı ile, parçalanan veri bloklarından bir kısmına erişilemiyor olunması, tüm verinin içerik açısından etkilenmesine neden olmakta, veri üzerinde yürütülen sorguların tamamlanamamasına neden olmaktadır. Bu durumun çözümü için, aynı veri bloğunun birden fazla uç nokta üzerinde saklanmasını ve sorgu yapan birimin, olası uç birimlerden en uygun olanını seçmesini sağlayan bir metot geliştirilebilir.

Çalışmalar sırasında tespit edilen önemli diğer bir husus ise, dağıtık yapılarda alt birimlerin işlemler açısından birbirlerini engellemeyecek şekilde çalışmasının önemli bir tasarım kararı olduğudur. Bu özelliğin gerçeklenmemesi durumunda, küme birimlerinden birisinin görevini öngörüldüğü şekilde tamamlayamaması durumunda, sistemin toplam başarımının olumsuz bir şekilde etkilediğidir.

Planlanan ikinci iyileştirme ise, hem yukarıdaki iki paragrafta belirtilen problemleri çözmek için, hem de sistem kullanım oranını yükseltmek amacı ile, verinin birden fazla uç birim üzerinde kopyalarının bulunmasını sağlamaktır. Ayrıca birinci

iyileştirme önerisinde belirtilen türde, dinamik olarak değişen paylaşılmış durum bilgisi sayesinde, hangi alt sorguların hangi uç birimlerden cevaplanabileceğinin ve bu birimlerin mevcut durumlarının ne olduğunun bilinmesi, aynı sorgunun farklı uç birimlerden birebir aynı şekilde cevaplanmasına olanak verecektir. Bunun başarılması, verinin birden fazla uç birimde bulunuyor olması dolayısı ile, hem yedeklilik ihtiyacını karşılayacak, hem de meşgul uç birimin cevap vermesi için beklenen süre, başka bir uç birim kullanılarak ortadan kaldırılacaktır. Böylece toplam cevaplama süresine artı katkı sağlanacaktır.

Yapılabilecek diğer bir geliştirme ise, sistem yeteneklerinin arttırılması adına, W3C tarafından yayınlanan SPARQL öneri dokümanlarındaki özelliklerden daha fazlasını uygulamaya dökmektir. Bunun gerçekleştirilmesi durumunda sistemin cevap verebildiği sorgu türü sayısı arttırılmış olacaktır. Örneğin, tez çalışması sırasında, LUBM sorguları göz önüne alındığı ve bu sorgularda, "ORDER BY, DISTINCT, OFFSET, LIMIT" gibi sorgu değiştiriciler (Modifiers) ve sonuç sıralayıcılar (Solution Sequencer) kullanılmadığı için, bunlar, gerçekleştirime dahil edilmemişlerdir. Ayrıca, benzer şekilde, "CONSTRUCT, DESCRIBE" sorgu türleri ve opsiyonel cevaplar elde edilmesi amacı ile kullanılan "OPTION" komutu ele alınmamıştır.

Bu tez çalışmasında zaman ölçümleri, DRS merkez uygulamasının çalıştırıldığı, küme uç birimi üzerinde yapılmaktadır. Sorgular DRS merkez uygulamasından gönderildiği an çalıştırılan bir sayıcı, cevaplar geri geldiği an durdurulmakta ve arada geçen zaman hesaplanmaktadır. Ancak bu şekilde yapılan bir ölçüm, alt işlemler sırasında geçen zamanların ayrı ayrı hesaplanması açısından, düşük çözünürlüğe sahip bir ölçüm yöntemidir. Bu yüzden, toplam zamanın ne kadarının veri arama ile, ne kadarının birleştirme işlemleri ile ve ne kadarının veri transferi ile geçtiği konusunda net bir bilgi elde edilememektedir. Böyle bir bilginin olması, alt işlemlerin iyileştirilmesi için geliştirilecek çözümlere ışık tutacaktır. Doğal olarak diğerlerine göre daha büyük zaman kayıplarına sebep olan adımların iyileştirilmesi, sistemin toplamda daha kısa sürede cevap oluşturmasını sağlayacaktır. Çözüm olarak zamanların, tüm alt birimlerde ayrı ayrı hesaplanması ve sorgu cevapları ile birlikte, toplanmak üzere merkez

uygulamaya gönderilmesi düşünülmüştür.

# KAYNAKLAR


[1] 4Store 0.9.1. http://www.4store.org/, 2009.

[2] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, Vienna,Austria, 2007.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[4] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: A vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.

[5] Apache Accumulo Project. http://accumulo.apache.org/, 2013.

[6] Apache Commons. http://commons.apache.org/, 2013.

[7] Apache Hadoop. http://hadoop.apache.org/, 2013.

[8] Apache Hadoop HDFS Documentation. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html, 2013.

[9] Apache HBase. http://http://hbase.apache.org/, 2013.

[10] Apache Jena Documentation. http://jena.apache.org/documentation/index.html, 2013.

[11] Apache Jena Fuseki. http://jena.apache.org/documentation/serving_data/, 2013.

[12] ARQ, A SPARQL Processor for Jena. http://openjena.org/ARQ/, 2013.

[13] D. Beckett. Rasqal RDF Parser Utility. http://librdf.org/rasqal/roqet.html, 2012.

[14] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American Magazine*, March 2008.

[15] C. Bizer. D2rq - treating non-rdf databases as virtual rdf graphs. In *In Proceedings of the 3rd International Semantic Web Conference (ISWC2004*, 2004.

[16] C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

[17] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 121–132, New York, NY, USA, 2013. ACM.

[18] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*, pages 54–68, Sardinia, Italia, 2002.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, and D. A. Wallach. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[20] C. Franke, S. Morin, A. Chebotko, J. Abraham, and P. Brazier. Distributed semantic web data management in hbase and mysql cluster. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 105–112, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2):158–182, July 2005.

[22] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *PSSS*, pages 43–48, 2003.

[23] S. Harris, N. Lamb, and N. Shadbolt. N.: 4store: The design and implementation of a clustered rdf store. In *In: Scalable Semantic Web Knowledge Base Systems - SSWS2009*, pages 94–109, 2009.

[24] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *of Lecture Notes in Computer Science*, pages 211–224. Springer, 2007.

[25] J. Hebeler, M. Fisher, R. Blace, and A. Perez-Lopez. *Semantic Web Programming*. Wiley, 2009.

[26] I. Herman. Eleven SPARQL 1.1 specifications are W3C recommendations. `http://www.w3.org/blog/SW/2013/03/21/eleven-sparql-1-1-specifications-are-w3c-recommendations/`, March 2013.

[27] METIS-Serial Graph Partitioning. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`, 2011.

[28] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[29] F. Inc. AllegroGraph 4.12. `http://www.franz.com/agraph/allegrograph/`, 2013.

[30] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[31] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Journal on Scientific Computing*, 20(1):359–392, 1999.

[32] V. Khadilkar, M. Kantarcioglu, B. Thuraisingham, and P. Castagna. Jena-hbase a distributed, scalable and efficient rdf triple store. In *International Semantic Web Conference*, Boston,USA, November 2012.

[33] C. Lam. *Hadoop In Action*. MANNING, 2011.

[34] B. McBrid. Jena: A semantic web toolkit. *IEEE Internet Computing*, pages 55–59, November 2002.

[35] Mulgara 2.1.13. `http://www.mulgara.org/download.html`, 2012.

[36] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19:91–113, 2010.

[37] OpenLink Virtuoso. `http://virtuoso.openlinksw.com/`, 2013.

[38] Sesame Framework for storage and querying of RDF data. `http://www.openrdf.org/`, 2013.

[39] A. Owens, A. Seaborne, and N. Gibbins. Clustered TDB: A Clustered Triple Store for Jena - ECS EPrints Repository. 2009.

[40] BigOWLIM. `http://www.ontotext.com/owlim`, 2013.

[41] R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: a scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, pages 4:1–4:8, New York, NY, USA, 2012. ACM.

[42] K. Rohloff and R. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. *International Workshop on Programming Support Innovations for Emerging Distributed Applications*, 2010.

[43] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[44] Apache Jena SDB. `http://jena.apache.org/documentation/sdb/`, 2013.

[45] Apache Jena TDB. `http://jena.apache.org/documentation/tdb/`, 2013.

[46] SWAT Projects - the Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/, 2012.

[47] J. Venner. *Pro Hadoop*. Apress, 2009.

[48] Resource Description Framework(RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, February 1999.

[49] OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/, February 2004.

[50] RDF Primer. W3C Recommendation. http://www.w3.org/TR/rdf-primer, 2004.

[51] SKOS Simple Knowledge Organization System Reference. http://www.w3.org/TR/2009/REC-skos-reference-20090818/, August 2009.

[52] A list of sparql implementations. http://www.w3.org/wiki/SparqlImplementations, 2013.

[53] SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/, 2013.

[54] T. White. *Hadoop, The Definite Guide*. O'Reilly, 2012.

[55] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *In Proceedings of SWDB'03, 1st International Workshop on Semantic Web and Databases, Co-located with VLDB 2003*, pages 131–150, 2003.

[56] L. Yu. *A Developer's Guide to the Semantic Web*. Springer-Verlag, 2011.

[57] V. Zhirnov, R. Cavin, J. Hutchby, and G. Bourianoff. Limits to binary logic swith scaling - a gedanken model. *Proocedings of the IEEE*, 91:1934–1939, Nov 2003.

# EKLER

# A. Verilerin Üretilmesi

Bu ekte deneyler sırasında kullanılan verinin LUBM veri üretici kullanılarak üretilişi açıklanmıştır.

## A.1 LUBM Veri Üretimi

Bu işlem için Ubuntu Linux kurulu bir bilgisayar kullanılmıştır. LUBM veri üreticinin adı UBA1.7'dir.

1. Veri üretici (LUBM Data Generator) dosyası http://swat.cse.lehigh.edu/projects/lubm/ web adresinden indirilebilir. Bu tez çalışmasının yazıldığı sırada mevcut olan en güncel sürüm UBA1.7 versiyonudur.
2. İndirilen Zip dosyası, veri üretimi yapılacak olan dizin içinde açılmalıdır.
3. Veri üretici yazılım, bir Java uygulaması olduğu için, kodun çalıştırılacağı dizinin "CLASSPATH" içinde olması gerekmektedir. Eğer, kullanılan sistemde sabit bir CLASSPATH çevresel değişkeni var ise, o değişkene "export" komutu aracılığı ile ";." ifadesi veya içinde bulunulan dizin/class dosyalarını bulunduran dizin eklenmelidir. Bu adım farklı işletim sistemlerinde farklı bir şekilde yürütülüyor olabilmektedir.
4. Programın çalıştırılması için diğer alternatif ise, uygulamanın "Classpath" parametresi ile çalıştırılmasıdır. Bunun için *Kurulum_Dizini/uba1.7/* içinde iken "java -classpath ./classes/ edu. lehigh.swat.bench.

uba.Generator -univ 1 -onto http://swat.cse.lehigh.edu/ onto/univ-bench.owl" komutu kullanılabilir. Bu programın çalıştırılması sonucu, bulunulan dizin içinde, OWL yapısında birçok veri dosyasının oluştuğu görülecektir. Verilen komut içindeki 1 rakamı verinin göreceli büyüklüğünü göstermektedir. 1 sadece tek üniversite için veri üretecektir. Veriyi büyütmek için bu sayının büyütülmesi gerekmektedir.

## A.2 Verinin RDF'e Dönüştürülmesi

OWL yapısında üretilen verilerin, N-Triple yapısına çevrilmesi için, RDF2RDF isimli ve http://www.l3s.de/ minack/rdf2rdf/ adresinden alınabilecek bir java uygulaması kullanılmaktadır. Bu uygulama ile veriyi dönüştürmek için, "java -jar rdf2rdf-1.0.1-2.3.1.jar INPUTFILE(S) OUTPUTFILE" ifadesine uygun bir şekilde çalıştırılması gerekmektedir. INPUT ve OUTPUT dosya tipleri çevrilecek veri türlerine göre seçilmelidir. Bu çalışmada OUTPUTFILE .nt ile biten dosyalar iken, INPUTFILE .owl ile biten üretilmiş verilerdir.

## A.3 Metis Kurulumu ve Kullanımı

METIS, kaynak kodlarından derlenerek kurulmaktadır. Yazılımın kaynak kodları, http:// glaros.dtc .umn.edu/gkhome /metis/metis/overview web adresinde bulunan sayfadan alınabilmektedir. Metis programını derlemek için yapılması gereken tek şey install.txt dokümanının içinde de yazıldığı gibi, indirilen kaynak kodlarının bulunduğu dizinde, önce "make config", daha sonra "make" komutunun çalıştırılmasıdır. Bu yüzden, GNU Cmake ve GCC programları bilgisayarda kurulu olmalıdır.

GKLib kütüphanesinin derlenmesi sırasında, içinde bulunulan dizine ait dizin yolu içinde, boşluk içeren bir dizin adı bulunması durumunda bir derleme hatası oluşmaktadır. Bu dizin ismi düzeltilince derleme işlemi başarı ile

tamamlanacaktır.

Derleme işlemi tamamlandığında, derleme işleminin yürütüldüğü dizin içinde oluşan, build/Linux-x86_64/programs dizini içinde çalıştırılabilir dosyalar oluşacaktır. Çizge parçalama (Graph Partitioning) işlemi için "gpmetis" uygulaması kullanılacaktır. Örneğin, "data" isimli bir RDF dosyasının 5 parçaya ayrılması için verilecek komut "gpmetis data out.txt 5" şeklindedir. Bu komutun çalıştırılması sonucunda, aynı dizin içinde out.txt isimli bir dosyanın oluştuğu görülecektir.

# B. LUBM Test Sorguları

Bu bölümde testler sırasında kullanılan LUBM sorguları verilmiştir. Sorgular, LUBM web sayfasından alınan orijinal halleri ile gösterilmiştir.

SPARQL B.1: LUBM Test Sorgusu 1

```
# Query1
# This query bears large input and high selectivity. It queries about
# just one class and one property and does not assume any
# hierarchy information or inference.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse
http://www.Department0.University0.edu/GraduateCourse0}
```

SPARQL B.2: LUBM Test Sorgusu 2

```
# Query2
# This query increases in complexity: 3 classes and 3 properties are
# involved. Additionally, there is a triangular pattern of
# relationships between the objects involved.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
```

```
{?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:University .
  ?Z rdf:type ub:Department .
  ?X ub:memberOf ?Z .
  ?Z ub:subOrganizationOf ?Y .
  ?X ub:undergraduateDegreeFrom ?Y}
```

SPARQL B.3: LUBM Test Sorgusu 3

```
# Query3
# This query is similar to Query 1 but class Publication has a
# wide hierarchy.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Publication .
  ?X ub:publicationAuthor
        http://www.Department0.University0.edu/AssistantProfessor0}
```

SPARQL B.4: LUBM Test Sorgusu 4

```
# Query4
# This query has small input and high selectivity. It assumes
# subClassOf relationship between Professor and its subclasses.
# Class Professor has a wide hierarchy. Another feature
# is that it queries about multiple properties of a single class.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y1, ?Y2, ?Y3
WHERE
{?X rdf:type ub:Professor .
  ?X ub:worksFor <http://www.Department0.University0.edu> .
  ?X ub:name ?Y1 .
  ?X ub:emailAddress ?Y2 .
  ?X ub:telephone ?Y3}
```

SPARQL B.5: LUBM Test Sorgusu 5

```
# Query5
# This query assumes subClassOf relationship between Person
# and its subclasses and subPropertyOf relationship between
# memberOf and its subproperties. Moreover, class Person
# features a deep and wide hierarchy.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Person .
  ?X ub:memberOf <http://www.Department0.University0.edu>}
```

SPARQL B.6: LUBM Test Sorgusu 6

```
# Query6
# This query queries about only one class. But it assumes
# both the explicit subClassOf relationship between
# UndergraduateStudent and Student and the implicit one
# between GraduateStudent and Student. In addition, it has large
# input and low selectivity.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X WHERE {?X rdf:type ub:Student}
```

SPARQL B.7: LUBM Test Sorgusu 7

```
# Query7
# This query is similar to Query 6 in terms of class Student
# but it increases in the number of classes and properties
# and its selectivity is high.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
```

```
SELECT ?X, ?Y
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Course .
  ?X ub:takesCourse ?Y .
  <http://www.Department0.University0.edu/AssociateProfessor0>,
        ub:teacherOf, ?Y}
```

SPARQL B.8: LUBM Test Sorgusu 8

```
# Query8
# This query is further more complex than Query 7
# by including one more property.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Department .
  ?X ub:memberOf ?Y .
  ?Y ub:subOrganizationOf <http://www.University0.edu> .
  ?X ub:emailAddress ?Z}
```

SPARQL B.9: LUBM Test Sorgusu 9

```
# Query9
# Besides the aforementioned features of class Student
# and the wide hierarchy of class Faculty, like
# Query 2, this query is characterized by the most
# classes and properties in the query set and there is
# a triangular pattern of relationships.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:Student .
```

```
  ?Y rdf:type ub:Faculty .
  ?Z rdf:type ub:Course .
  ?X ub:advisor ?Y .
  ?Y ub:teacherOf ?Z .
  ?X ub:takesCourse ?Z}
```

SPARQL B.10: LUBM Test Sorgusu 10

```
# Query10
# This query differs from Query 6, 7, 8 and 9 in that
# it only requires the (implicit) subClassOf relationship
# between GraduateStudent and Student, i.e., subClassOf
# relationship between UndergraduateStudent and Student
# does not add to the results.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Student .
  ?X ub:takesCourse
<http://www.Department0.University0.edu/GraduateCourse0>}
```

SPARQL B.11: LUBM Test Sorgusu 11

```
# Query11
# Query 11, 12 and 13 are intended to verify the presence
# of certain OWL reasoning capabilities in the system.
# In this query, property subOrganizationOf is defined
# as transitive. Since in the benchmark data, instances
# of ResearchGroup are stated as a sub-organization of a
# Department individual and the later suborganization of
# a University individual, inference about the subOrgnizationOf
# relationship between instances of ResearchGroup and University
# is required to answer this query. Additionally, its input is small.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
```

```
SELECT ?X
WHERE
{?X rdf:type ub:ResearchGroup .
  ?X ub:subOrganizationOf <http://www.University0.edu>}
```

SPARQL B.12: LUBM Test Sorgusu 12

```
# Query12
# The benchmark data do not produce any instances of
# class Chair. Instead, each Department individual is
# linked to the chair professor of that department by
# property headOf. Hence this query requires realization,
# i.e., inference that that professor is an instance of
# class Chair because he or she is the head of a
# department. Input of this query is small as well.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y
WHERE
{?X rdf:type ub:Chair .
  ?Y rdf:type ub:Department .
  ?X ub:worksFor ?Y .
  ?Y ub:subOrganizationOf <http://www.University0.edu>}
```

SPARQL B.13: LUBM Test Sorgusu 13

```
# Query13
# Property hasAlumnus is defined in the benchmark ontology
# as the inverse of property degreeFrom, which has three
# subproperties: undergraduateDegreeFrom, mastersDegreeFrom,
# and doctoralDegreeFrom. The benchmark data state a person as
# an alumnus of a university using one of these three subproperties
# instead of hasAlumnus. Therefore, this query assumes
# subPropertyOf relationships between degreeFrom and its
# subproperties, and also requires inference about inverseOf.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Person .
  <http://www.University0.edu> ub:hasAlumnus ?X}
```

SPARQL B.14: LUBM Test Sorgusu 14

```
# Query14
# This query is the simplest in the test set. This query
# represents those with large input and low selectivity and
# does not assume any hierarchy information or inference.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {?X rdf:type ub:UndergraduateStudent}
```

# C. LUBM Verileri İçin Kullanılan Ontoloji

LUBM veri üretme yazılımı olan UBA1.7, test sorgularının cevaplanabilmesi için Univ-bechmark.owl isimli bir ontoloji kullanmaktadır. Bu ontoloji, bir üniversite ve bileşenlerini modellemektedir. Ontolojinin küçük bir kısmı aşağıda örnek olarak verilmiştir.

XML C.1: LUBM univ-bech.owl ontolojisi

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF
  xmlns = "http://swat.cse.lehigh.edu/onto/univ-bench.owl#"
  xml:base = "http://swat.cse.lehigh.edu/onto/univ-bench.owl"
  xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
>

<owl:Ontology rdf:about="">
  <rdfs:comment>An university ontology for benchmark tests</rdfs:comment>
  <rdfs:label>Univ-bench Ontology</rdfs:label>
  <owl:versionInfo>univ-bench-ontology-owl, ver April 1,
      2004</owl:versionInfo>
</owl:Ontology>

<owl:Class rdf:ID="AdministrativeStaff">
```

```
  <rdfs:label>administrative staff worker</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Employee" />
</owl:Class>

<owl:Class rdf:ID="Article">
  <rdfs:label>article</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Publication" />
</owl:Class>

<owl:Class rdf:ID="AssistantProfessor">
  <rdfs:label>assistant professor</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Professor" />
</owl:Class>

<owl:Class rdf:ID="AssociateProfessor">
  <rdfs:label>associate professor</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Professor" />
</owl:Class>

<owl:Class rdf:ID="Book">
  <rdfs:label>book</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Publication" />
</owl:Class>

<owl:Class rdf:ID="Chair">
  <rdfs:label>chair</rdfs:label>
  <owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Person" />
  <owl:Restriction>
  <owl:onProperty rdf:resource="#headOf" />
  <owl:someValuesFrom>
  <owl:Class rdf:about="#Department" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
```

```
52   <rdfs:subClassOf rdf:resource="#Professor" />
53 </owl:Class>
54
55 <owl:Class rdf:ID="ClericalStaff">
56   <rdfs:label>clerical staff worker</rdfs:label>
57   <rdfs:subClassOf rdf:resource="#AdministrativeStaff" />
58 </owl:Class>
59
60 <owl:Class rdf:ID="College">
61   <rdfs:label>school</rdfs:label>
62   <rdfs:subClassOf rdf:resource="#Organization" />
63 </owl:Class>
64
65 <owl:Class rdf:ID="ConferencePaper">
66   <rdfs:label>conference paper</rdfs:label>
67   <rdfs:subClassOf rdf:resource="#Article" />
68 </owl:Class>
69
70 <owl:Class rdf:ID="Course">
71   <rdfs:label>teaching course</rdfs:label>
72   <rdfs:subClassOf rdf:resource="#Work" />
73 </owl:Class>
74
75 <owl:Class rdf:ID="Dean">
76   <rdfs:label>dean</rdfs:label>
77   <owl:intersectionOf rdf:parseType="Collection">
78   <owl:Restriction>
79   <owl:onProperty rdf:resource="#headOf" />
80   <owl:someValuesFrom>
81   <owl:Class rdf:about="#College" />
82   </owl:someValuesFrom>
83   </owl:Restriction>
84   </owl:intersectionOf>
```

# D. Örnek Çıktılar

## D.1 RDF Üçlü Bileşen Sayıları

Bir Hadoop Map-Reduce uygulaması çalıştırılarak oluşturulan ve RDF dosyaları içerisindeki üçlü bileşenlerinin sayısını gösteren çıktının bir kısmı aşağıda verilmiştir. Bu dosyanın oluşturulması için, RDF dosyaları HDFS üzerine aktarıldıktan sonra, dosyaları satır satır okuyan, okunan her satırı bileşenlerine ayırdıktan sonra, bileşen ismine göre bir hash tablosuna anahtar olarak ekleyen ve aynı anahtara her rastlanıldığında, anahtar için tutulan değeri bir arttıran Map-Reduce uygulaması yazılmıştır. Bu uygulamanın ürettiği düz metin çıktı dosyasının bir bölümü Hadoop-D.1 içinde örnek olarak verilmiştir. Burada her satır bir üçlü bileşenini ve o bileşenin RDF dosyalarında toplam kaç kere geçtiğini göstermektedir. Örneğin, "GraduateCourse0" bileşeni toplam 15 kere görülmüş[1].

Hadoop D.1: RDF dosyaları içerisindeki üçlülere ait bileşenlerin, bir Hadoop Map-Reduce uygulaması ile sayılması sonucunda oluşan dosyanın örnek bir bölümü

```
1 "AssistantProfessor0" 15
2 "AssistantProfessor0@Department0.University0.edu" 1
3 "AssistantProfessor1" 15
4 "AssistantProfessor10@Department10.University0.edu" 1
5 "Course0"     15
6 "Course1"     15
7 "Course12"    15
```

[1]Bu örnek için kullanılan RDF dosyasının boyutu 1GByte'tır. Veri boyutu büyüdükçe bileşenlerin görülme sayıları artacaktır.

```
"FullProfessor2@Department7.University0.edu" 1
"FullProfessor3"      15
"FullProfessor7"      9
"FullProfessor7@Department0.University0.edu" 1
"GraduateCourse0"     15
"GraduateStudent105@Department9.University0.edu"   1
"GraduateStudent106"  14
"GraduateStudent107@Department11.University0.edu"  1
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#FullProfessor> 125
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateCourse>
    799
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>
    1874
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#ResearchAssistant>
    547
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#ResearchGroup> 224
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#UndergraduateStudent>
    5916
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse> 21489
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#teacherOf> 1627
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#worksFor> 540
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl> 15
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  20674
<http://www.w3.org/2002/07/owl#Ontology>    15
<http://www.w3.org/2002/07/owl#imports> 15
```

# E. Apache Hadoop Kurulumu

Bir Hadoop kümesinde, yaptığı işe göre özelleşmiş farklı sayıda ve isimde düğümler (cluster node) bulunabilir. Bu düğümler genel olarak yönetici (master) ve işlemci (slave) olarak iki gruba ayrılırlar. Yönetici düğümler "jobtracker" ve "namenode" iken, işlemci düğümler "datanode" ve "tasktracker" düğümleridir. Kümenin kurulum biçimine göre bu servis uygulamalarının (daemon) hepsi aynı fiziksel bilgisayarın üzerinde olabileceği gibi (Single Node Cluster Setup), farklı bilgisayarlar üzerinde de olabilirler. Bu dokümanda ikinci durum anlatılacaktır. Tipik bir Hadoop küme kurulumunda "namenode" ve "jobtracker" ayrı makinalarda kurulurken, "datanode" ve "tasktracker" aynı bilgisayar üzeride kurulurlar. Kümede "datanode" sayısı yapılacak işlemin ölçeğine göre ayarlanır.

Bu dokümanda kullanılacak küme için bilgisayar adları ve IP'leri aşağıdaki gibidir:

Hadoop E.1: Hadoop Kümesi

```
1    Job Tracker : jobtracker :192.168.122.29
2    Name Node : namenode :192.168.122.30
3    Data Node1 : datanode1 :192.168.122.31
4    Data Node2 : datanode2 :192.168.122.32
5    Data Node3 : datanode3 :192.168.122.33
6    Data Node4 : datanode4 :192.168.122.34
```

## E.1 Genel Tanımlamalar

1. **Namenode** "Namenode" isim uzayını (namespace), dosya sistemi üst verisini (metadata) ve erişim kontrolünü yönetir. Her kümede sadece bir tane "namenode" bulunur. Ancak "namenode" kümenin çalışması açısından çok kritik bilgiler içerdiği için genellikle birebir bir kopyası olacak şekilde yedeklenir.

2. **SecondaryNameNode** SecondaryNameNode, sistemin hata toleransını arttırmak için, "namenode" üzerindeki bilgiden periyodik olarak kontrol noktaları (checkpointler) kaydeder. Bu sayede "namenode" düğümün arıza veya hata yapması durumunda dahi, kümenin çalışmaya devam edebilmesini sağlar. Secondary Namenode da kümede bir tane bulunur.

3. **Jobtracker** "Jobtracker" sistemdeki işleri (task) yöneten makinadır. İşlerin işlemci düğümlere dağıtılması işlemini yönetir. Her kümede sadece bir tane bulunur.

4. **Datanode** Dosya sistemini oluşturan ve veriyi doğrudan depolayan bilgisayarlardır. Ancak dosya sistemindeki dosyaların metadata bilgileri "namenode" üzerinde saklanır. Bir kümede birden fazla sayıda "datanode" bulunabilir. Her "datanode" kendi üzerinde bulunan storage alanını yönetir. Bu saklama alanlarında file sistem üzerinde bulunan dosyaların bloklarının bir kısmının veya tamamının bir kopyası bulunur. Eğer kümede sadece bir adet "datanode" bulunursa filesystem yedekliliği yapılamaz.

## E.2 Ortak Konfigürasyonlar

Bu bölümde kümede tüm makinalarda yapılması gereken konfigürasyon adımları anlatılmaktadır.

1. Tüm düğümler, genel konfigürasyonları aynı olacak şekilde kurulur, hepsine kalıcı IP verilir.

2. Tüm node isimleri için DNS kayıtları oluşturulup, tüm bilgisayarlar için aynı DNS sunucusu tanımlanır(DNS sunucusu kurmaya gerek olmayacak kadar az sayıda bilgisayar var ise /etc/hosts dosyası içinde tüm düğümler tanımlanır).

3. Tüm makinalara Java JDK kurulumu yapılır (hepsinde aynı versiyon olması(JDK 1.6.x) ve Sun dağıtımı olması önemli).

4. Tüm makinalarda SSH kurulu olmalı ve hepsi SSHD çalıştırıyor olmalıdır (Bu Hadoop betiklerinin diğer düğümler üzerinde komut çalıştırabilmesi için gereklidir.)

5. Tüm makinalara 'data' isimli bir dizin açılıp, internetten edilen hadoop.tar.gz dosyası bu alana açılır.

6. Bütün makinalarda açılan tar dosyasının içinde bulunan conf/hadoop-env.sh dosyası içinde JAVA_HOME değeri java kurulumunun olduğu dizini gösterecek şekilde kayıt edilir.

7. $HADOOP_HOME/conf/core-site.xml dosyası içine aşağıdaki ifade eklenir. Bu konfigürasyon kümedeki tüm makinalarda (namenode + jobtracker + datanodes) aynı olmalıdır. Kümenin kullandığı dosya sistemi yönetimini hangi makinanın yaptığı bilgisini depolar. Ayrıca Hadoop tarafından kullanılan temporary dizinin neresi olduğu bilgisini saklar.

Hadoop E.2: Hadoop core-site.xml

```
1  <property>
2    <name>fs.default.name</name>
3        <value>hdfs://namenode:8020</value>
4  </property>
5  <property>
6        <name>hadoop.tmp.dir</name>
7        <value>/data/hadooptmp</value>
8  <description>Temporary directories.</description>
9  </property>
```

8. Tüm makinalarda ortak olması gereken diğer bir konfigürasyon dosyası da $HADOOP_HOME/conf/mapred-site.xml dosyasıdır. Bu dosyada map-reduce işlerini yönetecek olan bilgisayarın hangisi olduğu ve map-reduce işleri hakkında temel konfigürasyonlar bilgileri verilmektedir.

Hadoop E.3: Hadoop mapred-site.xml

```
1
2   <property>
3         <name>mapred.job.tracker</name>
4         <value>jobtracker:8021</value>
5         <final>true</final>
6   </property>
7   <property>
8         <name>mapred.local.dir</name>
9         <value>/data/mapred/local</value>
10        <final>true</final>
11  </property>
12  <property>
13        <name>mapred.system.dir</name>
14        <value>/tmp/hadoop/mapred/system</value>
15        <final>true</final>
16  </property>
17
18  <property>
19    <name>mapred.tasktracker.map.tasks.maximum</name>
20    <value>4</value>
21    <final>true</final>
22  </property>
23  <property>
24        <name>mapred.tasktracker.reduce.tasks.maximum</name>
25        <value>1</value>
26        <final>true</final>
27  </property>
28  <property>
29        <name>mapred.child.java.opts</name>
```

```
30        <value>-Xmx400m</value>
31        <!-- Not marked as final so jobs can include JVM debugging
              options -->
32    </property>
```

9. Tüm makinalarda $HADOOP_HOME/conf/hdfs-site.xml dosyası üzerinde de ortak bazı değişiklikler yapılacaktır. Bu dosya HDFS file sistemi ile ilgili bazı konfigürasyon parametrelerini içermektedir. Burada belirtilen dizinleri Hadoop kendisi üretecektir, kullanıcı tarafından üretilmemelidir.

Hadoop E.4: Hadoop hdfs-site.xml

```
1   <property>
2         <name>dfs.name.dir</name>
3         <value>/data/hdfs/name</value>
4         <final>true</final>
5   </property>
6   <property>
7         <name>dfs.data.dir</name>
8         <value>/data/hdfs/data</value>
9         <final>true</final>
10  </property>
11  <property>
12        <name>fs.checkpoint.dir</name>
13        <value>/data/hdfs/namesecondary</value>
14        <final>true</final>
15  </property>
```

10. $HADOOP_HOME/conf/slaves dosyası içerisine sistemde bulunan tüm "datanode" düğümlerinin isimleri, her biri bir satıra gelecek şekilde girilir. Bu dosya tüm düğümlerde aynı olmalıdır.

Hadoop E.5: Hadoop slaves dosyası

```
1   datanode1
2   datanode2
```

```
3 datanode3
4 datanode4
```

## E.3 Birime Özgü Konfigürasyonlar

Birime özgü konfigürasyon bulunmamaktadır.

## E.4 Map-Reduce Test Kodu

Hadoop ile birlikte dağıtılan örnek uygulamaların bulunduğu Jar dosyası içinde PI sayısının hesaplanması amacı ile yazılmış olan bir Map-Reduce uygulaması bulunmaktadır. Bu uygulama aşağıdaki komut ile çalıştırılarak, Hadoop kümesinin düzgün kurulup kurulmadığı test edilebilir.

bin/hadoop jar Hadoop-0.18.0-examples.jar pi 10 1000000

# ÖZGEÇMİŞ

### Kişisel Bilgiler


| | |
|---|---|
| Soyadı, Adı | : EROĞLU, Özgür |
| Uyruğu | : T.C. |
| Doğum tarihi ve yeri | : 04.07.1976 Erzincan |
| Medeni hali | : Evli |
| Telefon | : 0 536 257 63 80 |
| Faks | : |
| e-mail | : oeroglu@.etu.edu.tr |


### Eğitim

| Derece | Eğitim Birimi | Mezuniyet Tarihi |
|---|---|---|
| Yüksek Lisans | TOBB ETÜ -Bilgisayar Mühendisliği Bölümü | 2013 |
| Lisans | Orta Doğu Teknik Üniversitesi-FİZİK Bölümü | 2002 |

### İş Deneyimi

| Yıl | Yer | Görev |
|---|---|---|
| 2012-2013 | Onur A.Ş. | Lider Yazılım Mühendisi |
| 2011-2011 | Critical Factor | Yazılım Mühendisi |
| 2008-2011 | Bilgi GIS Ltd. | Bilişim Altyapısı ve Güvenlik Yöneticisi |
| 2005-2008 | Selex Komünikasyon A.Ş. | IS Engineer |
| 2002-2003 | NanoManyetik Bilimsel Cihazlar | Ar-Ge Personeli |

### Yabancı Dil

İngilizce (Çok iyi)

### Yayınlar