KOCAELİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

YÜKSEK LİSANS TEZİ

HADOOP TABANLI BÜYÜK ÖLÇEKLİ GÖRÜNTÜ İŞLEME ALTYAPISI

İLGİNÇ DEMİR

KOCAELİ 2012

KOCAELİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

YÜKSEK LİSANS TEZİ

HADOOP TABANLI BÜYÜK ÖLÇEKLİ GÖRÜNTÜ İŞLEME ALTYAPISI

İLGİNÇ DEMİR

Yrd.Doç.Dr.Ahmet SAYAR
Danışman, Kocaeli Üniv. .....

Prof.Dr. Yaşar BECERİKLİ
Jüri Üyesi, Kocaeli Üniv. .....

Doç.Dr. Erdoğan SEVİLGEN
Jüri Üyesi, GYTE .....

Tezin Savunulduğu Tarih: 28.06.2012

ÖNZÖZ VE TEŞEKKÜR

Son yıllarda çoklu-medya (multimedia) kullanımı, özellikle internetin de yaygınlaşmasıyla çok büyük boyutlara ulaşmıştır. Sosyal paylaşım siteleri ve video paylaşım amaçlı kurulmuş sunucularda tutulan verinin güvenli bir şekilde kaydedilmesi ve bu veriye hızlı bir şekilde erişilerek verilerin sınıflandırma ve arama amaçlı olarak işlenmesi gerekmektedir. Dağıtık dosya sistemleri üzerinde paralel hesaplama yöntemleri bu amaçla kullanılmaya başlamıştır. Bu alanda kullanılan en güncel teknolojiler ve altyapılar üzerinde çözüm üretmek benim için çok heyecan verici ve teşvik edici olmuştur.

Yüksek lisans eğitimim süresince değerli birikimlerini benimle paylaşan, tezimin her aşamasında sorunlarımı dinleyerek, çalışmalarıma yön veren ve yoğun akademik yaşamında değerli zamanını her türlü problemimi çözmeye ayıran tez danışmanım saygıdeğer hocam Yrd. Doç. Dr. Ahmet SAYAR'a teşekkürlerimi sunarım.

Bugünlere gelmemi sağlayan anneme, babama ve bana her konuda katlanarak çalışmalarımda manevi desteğini eksik etmeyen değerli eşim Vijdan'a ve kendisiyle tez çalışmam dolayısıyla bazen ilgilenemediğim oğlumuz İhsan'a saygı, sevgi ve sonsuz teşekkürler.

Haziran – 2012 İlginç Demir

İÇİNDEKİLER

ŞEKİLLER DİZİNİ

TABLOLAR DİZİNİ

SİMGELER DİZİNİ VE KISALTMALAR

HDFS	: Hadoop Distributed File System (Hadoop Dağıtık Dosya Sistemi)
GFS	: Google File System (Google Dosya Sistemi)
DFS	: Distributed File System (Dağıtık Dosya Sistemi)
JNI	: Java Native Interface (Java Yerel Arayüzü)
RPC	: Remote Procedure Call (Uzaktan Erişimle Fonksiyon Çağırma)
UML	: Unified Modeling Language (Birleşik Modelleme Dili)
JVM	: Java Virtual Machine (Java Sanal Makinası)
XML	: Extensible Markup Language (Genişletilebilir İşaretleme Dili)
JAR	: Java Archive (Java Arşiv Dosyası)

HADOOP TABANLI BÜYÜK ÖLÇEKLİ GÖRÜNTÜ İŞLEME ALTYAPISI

ÖZET

Hadoop Dağıtık Dosya Sistemi (Hadoop Distributed File System (HDFS)) büyük boyutlu veriyi saklama ve işlemede yaygın olarak kullanılmaktadır. HDFS paralel işleme için MapReduce programlama modelini kullanır.

Bu tez ile sunulan çalışma kapsamında MapReduce modeli ile görüntü dosyalarının işlenebilmesi için yeni bir Hadoop eklentisi (plugin) geliştirilmiştir. Eklenti, görüntü ile ilgili girdi ve çıktı dosya formatları ve girdi dosyalarından kayıtları oluşturan yeni sınıfları içerir. HDFS özellikle az sayıda büyük boyutlu dosyalarla çalışması için tasarlanmıştır. Dolayısıyla, önerilen teknik, HDFS'de büyük miktarda küçük boyutlu görüntü dosyası kullanımından kaynaklanan performans kayıplarını önlemek için, imgelerin birleştirilerek büyük boyutlu dosyalara dönüştürülmesini temel almıştır. Böylelikle, her bir işleyici çok sayıda imgeyi tek çalışma döngüsünde işleyebilir hale gelir.

Önerilen tekniğin etkinliği dağıtık görüntü dosyaları üzerinde yüz saptama (face detection) uygulama senaryosu ile kanıtlanmıştır. Başarısı kanıtlanmış olan bu teknikler araştırmacılar ve uygulama geliştiriciler için bu konuda referans olma niteliği taşımaktadır.

Anahtar Kelimeler: Dağıtık Dosya Sistemleri, Eşle/İçerik (MapReduce), Hadoop, OpenCV, Yüz saptama

HADOOP BASED LARGE SCALE IMAGE PROCESSING INFRASTRUCTURE

ABSTRACT

Hadoop Distributed File System (HDFS) is widely used in large-scale data storage and processing. HDFS uses MapReduce programming model for parallel processing.

A novel Hadoop plugin to process image files with MapReduce model is developed in the work presented in this thesis. The plugin introduces image related I/O formats and novel classes for creating records from input files. HDFS is especially designed to work with small number of large size files. Therefore, the proposed technique is based on merging multiple small size files into one large file to prevent the performance loss stemming from working with large number of small size files. In that way, each task becomes capable of processing multiple images in a single run cycle.

The effectiveness of the proposed technique is proven by an application scenario for face detection on distributed image files. This successfull technique and developed plugin may become a reference for researchers and developers in this field.

Keywords: Distributed File Systems, MapReduce, Hadoop, OpenCV, Face Detection

GİRİŞ

Son on yılda, çoklu-medya (multi-medya) kullanımı özellikle internetin gelişmesine paralel olarak çok hızlı bir şekilde artmıştır. Flicker, Youtube ve Facebook gibi sosyal paylaşım sitelerinde tutulan çoklu-medya verileri çok büyük boyutlara ulaşmıştır. Arama motorları da günümüzde çoklu-medya arama özelliği sunduklarından, bu boyutta görüntü verilerinin kaydedilmesini ve işlenmesini yönetmek zorundadırlar.

Bu kadar büyük çoklu-medya verisinin tutulması ve paralel olarak işlenmesi için dağıtık sistemler kullanılmaktadır. Dağıtık sistemin yeni eklenen verilerle güvenli bir şekilde büyüyebilmesi ve bu veriler üzerinde paralel işleme yapılabilmesine olanak sağlaması gerekmektedir. Görüntülerin sınıflandırılabilmesi ve içeriğinde çeşitli aramalar yapılabilmesi için görüntü işleme algoritmalarının da bu dağıtık sistem içerisinde hızlı ve paralel bir şekilde çalıştırılabilir olması gerekmektedir.

Hadoop Dağıtık Dosya Sistemi (Hadoop Distributed File System (HDFS)) [1] büyük boyutlu verilerin dağıtık olarak kaydedilmesi ve işlenmesi için kullanılan sistemlerden biridir. Yaygın olarak; Yahoo gibi büyük arama motorlarında dağıtık olarak kaydedilen internet arşiv dosyaları üzerinde metin araması gibi işlemlerde kullanılır.

HDFS açık kaynak kodlu bir proje olarak geliştirilmiştir. HDFS veri işlemede Google Dosya Sistemi'nde (Google File System (GFS)) [2] kullanılan MapReduce [3] programlama yöntemini temel almıştır. Bu yöntem büyük ölçekli veriler üzerinde hesaplamanın paralelleştirilmesini çok etkin bir hale getirmiştir.

Çalıştırılacak algoritmanın çalıştığı düğüm (node) ile kullanacağı verinin farklı makinelerde olması dağıtık sistemlerde paralel işlemede yavaşlığa neden olan bir durumdur. HDFS'de işleyicinin (task) çalışacağı düğüm ile işleyeceği veri setinin bulunduğu düğümün yakın olması hedeflenmiştir. Veri yerelliği (data locality) denen bu durum paralel işleme hızını artırmaktadır [4].

HDFS özellikle büyük dosyaların kaydedilmesi ve işlenmesi için özelleşmiştir ve küçük boyutlu çok sayıda dosya ile kullanımında önemli performans kaybı olmaktadır. HDFS'de dosya sistemini yöneten NameNode yazılımı, dosya sayısı arttıkça tuttuğu dosya tanımlayıcı (metadata) verisi artar ve dolayısıyla hafıza kullanımını artırır, bu da yavaşlığa neden olur [5]. Ayrıca verilerin işlenmesi için oluşturulacak olan işleyici sayısı artar ve HDFS'de işlerin yönetilmesinden sorumlu olan JobTracker ve TaskTracker yazılımlarında ciddi yavaşlama olduğu görülür. Görev başlatma ve işleyicilerin sıralanması (schedule) performansı düşer [4]. Bu tez kapsamında hazırlanan imgeleri birleştirerek toplu olarak kaydetme ve işleme tekniği uygulaması buna çözüm olarak geliştirilmiştir.

Bu çalışmada, Hadoop içinde imge dosyalarını okuyup işleme ve çıktıları tekrar HDFS'ye istenen formatta kaydetmeyi sağlayacak girdi ve çıktı formatları geliştirildi. Bunun için görüntü işlemeye yönelik özelleşmiş yeni bir veri saklama (storage) ve kayıt oluşturma (record generation) eklentisi (plugin) geliştirildi. İkili (binary) dosya tipinde olan görüntü dosyalarının bir bütün olarak işlenebilmesi için bir Hadoop girdi dosyası formatı (ImageFileInputFormat) ve girdi dosyası kayıt okuyucusu sınıfı (ImageFileRecordReader) geliştirildi.

İmge işlemede kullanılacak algoritma olarak yüz saptama algoritması seçildi ve HDFS'de paralel yüz saptama ve saptanan yüzlerin kaydedilmesi için izlenen yöntemler açıklandı. Bu kapsamda, imgelerdeki yüzlerin saptanabilmesi için OpenCV [6] kütüphanesinde tanımlı Haar peş peşe sınıflayıcı yüz saptama algoritması (Haar Cascade Face Detector) kullanıldı. C++ dilinde yazılan bu kütüphane Java Native Interface (JNI) ile eklenti içerisine entegrasyonu yapıldı. Bu kütüphane ile saptanan yüzlerin imge olarak HDFS'ye kaydı sağlandı.

Her görüntü için bir Map işleyici çalıştırmak yeterli bir performans sağlamadığı bilindiğinden [5], imgeleri birleştirilerek, HDFS'nin daha performanslı çalıştığı bilinen, Hadoop'ta bulunan SequenceFile [7] dosya formatındaki dosyalara dönüştürme ve daha sonra bu dosyayı girdi olarak kullanarak imge işleme tekniği uygulandı ve eklenti ile tümleştirme yapıldı.

İstenen imge formatında ve isminde HDFS'ye kayıt sağlayabilmek için çıkış dosyası formatı (ImageFileOutputFormat) geliştirildi. Hazırlanan görüntü işleme görevi

çalıştırılarak farklı boyutlardaki görüntü kümeleri için sonuçlar analiz edildi. Geliştirilen eklentinin ve imge işleme görevlerinin performansı karşılaştırılarak sonuçlar sergilendi.

1. GENEL BİLGİLER

1.1. Tez Çalışmasının Amacı ve Başlatılma Sebepleri

Günümüzde özellikle internet üzerinde paylaşılan verinin kontrolü, sınıflandırılması ve güvenliği çok önemli bir ihtiyaç haline gelmiştir. Eskiden metin tabanlı olarak hazırlanan pek çok web sayfasında artık sadece resim ve videolar paylaşılarak insanlar duygularını başkalarıyla paylaşır olmuştur. Sunucularda tüm bu verinin güvenli bir şekilde tutulması, bu verilere hızlı bir şekilde erişilmesi için dağıtık dosya sistemleri kullanılmaya başlamıştır.

Hadoop'un bu alanda en çok tercih edilen bir dağıtık dosya sistemi (Distributed File System (DFS)) olması nedeniyle özellikle üzerinde çalışmak ciddi anlamda teşvik edici olmuştur. Yahoo, Facebook gibi web sunucularında altyapı olarak HDFS kullanılıyor olması da bu çalışmayı yapmada ayrıca teşvik edici olmuştur. Ülkemizde de bu boyutta sosyal paylaşım siteleri kurulmakta ve bu siteler özellikle görüntü işleme noktasında altyapı arayışına girebilmektedirler. Telekom sektöründe de kullanıcı verilerinin kütüklenmesi ve bu kütükler üzerinde işlem yapılabilmesi gerekebilmektedir. Bu nedenlerden dolayı ulusal projelerde kullanılabilirliği açısından da bu çalışma önemlidir.

HDFS yapısı ve çalışma şekli açısından küçük boyutlu görüntü verileri ile çalışma noktasında çok da başarılı olmayıp daha ziyade büyük boyutlu bölütlenebilen dosyalar ile çalışmaktadır. İmgelerin genelde küçük boyutlu dosyalar olması nedeniyle, Hadoop ile görüntü işleme yapılabilmesinin önünün açılmasında ve bu alanda çalışanlara bir yol haritası olması açısından bu çalışmanın yapılması faydalı olmuştur.

Gerek görsel veritabanları üzerinde aramalarda, gerek veri madenciliğinde ve gerekse internet sunucularının altyapılarının büyük çoklu-medya verilerini tutabilecek ve yönetebilecek şekilde hazırlanmasında bu çalışmadan faydalanılabileceği düşünülmektedir. Tüm bu sebeplerden dolay bu çalışma başlatılmış ve bu noktaya getirilmiştir.

1.2. Tez Çalışmasının Katkıları

Tez çalışması ile ulaşılmak istenen asıl amaç: Hadoop kullanarak dağıtık olarak büyük ölçekli görüntü işlemek için bir altyapı hazırlamaktır. Bu amaca ulaşmak için öncelikle; Hadoop ile görüntü işlemeyi dosya sistemi bazında mümkün kılan dosya formatları geliştirilmiştir. HDFS'den okunan imgelerin MapReduce iş parçacıklarına veri olarak girdisini sağlamak için kayıt okuyucu sınıfları geliştirilmiştir.

MapReduce iş parçacıkları ile imgelerde yüz saptama işleminin yapılabilmesi için OpenCV kütüphanesinin Java Native Interface (JNI) arayüzü ile entegrasyonu yapılmıştır. OpenCV kütüphanesi birçok görüntü işleme algoritmasını içerdiği için, ileride başka imge işleme görevlerini Hadoop üzerinde çalıştırabilmek açısından bu çalışma çok faydalı olacaktır. Saptanan yüzlerin HDFS'ye istenen formatta kaydını yapabilmek için çıktı formatı geliştirilmiştir. Bu çıktı formatı ile Hadoop'un görev çıktısı oluşturma yapısına esneklik kazandırılmış ve her tür görev çıktısının HDFS'ye istenen formatta yazılabilmesi sağlanmıştır. Bu girdi ve çıktı formatları ile ileride farklı çoklu-medya formatlarını işlemek de mümkün olabilecektir.

Geliştirilen bu sınıfları kullanarak imgelerin her birini tek bir işleyicide işleme tekniği ve imgeleri birleştirerek büyük boyutlu dosyaya dönüştürüp işleme tekniği geliştirilmiştir. Bu teknikler ile Hadoop'ta küçük boyutlu dosyaları kaydetme ve işleme yavaşlığı problemine imge işleme görevi kapsamında çözüm önerilmiştir.

1.3. Tez Düzeni

Tez çalışmaları, beş ana bölümde sunulmaktadır; Bölüm 2'de HDFS ve MapReduce ile ilgili daha önce yapılmış çalışmalar hakkında bilgiler sunulmuştur. Bölüm 3 Temel Anlatımlar bölümünde MapReduce programlama yöntemi ve HDFS anlatılmıştır. Bölüm 4'te Hadoop için geliştirilen eklentinin mimarisi ve yapısı anlatılmıştır. Ayrıca HDFS'de imge işleme amaçlı geliştirilen teknikler sunulmuştur Bölüm 5'te yüz saptama durum çalışması özelinde, geliştirilen eklenti kullanılarak hazırlanan görevlerin çalıştırılması açıklanmıştır. Yapılan tez çalışmasının sonuçları ve katkıları Bölüm 6'da değerlendirilmektedir.

2. İLGİLİ ÇALIŞMALAR

S. Ghemawat ve H. Gobioff [2] Google File System (GFS)'in altyapısını, sistemde çok büyük miktarda verinin nasıl tutulduğunu ve bunun Google dağıtık dosya sistemi tarafından nasıl yönetildiğini açıklamışlardır. Verilerin işlenebilmesi için GFS'de MapReduce yönteminin nasıl kullanıldığının sunumunu yapmışlardır. Bu çalışma DFS alanında somut örnek teşkil eden GFS'i açıklaması nedeniyle en çok referans verilen çalışmalardan biridir.

J. Dean ve S. Ghemawat [3] dağıtık sistemler üzerindeki veriyi paralel olarak işlemek için tanımladıkları MapReduce yöntemini açıklamışlardır. Bunu yaparken Map ve Reduce aşamalarının çıktılarının dağıtık sistemde nasıl kaydedildiğini ve bu çıktıların birleştirilme aşamalarını da açıklamışlardır. Google dosya sistemi (GFS) [2] üzerinde MapReduce programlama modelinin nasıl işlediğini anlatmışlardır. Özellikle veri yerelliği (Data Locality) prensibinin nasıl uygulandığı ve veri ile işleyicinin aynı düğümde olmasının MapReduce açısından önemi açıklanmıştır. MapReduce mimarisi ile kurulan sistemdeki yük dengelemeyi (Load Balancing) sağlamak için Map işleyici ve Reduce işleyicilerin sayısının nasıl belirlenmesi gerektiğine dair yöntemleri anlatmışlardır. Bu tez açısından ayrıca kullanıcı ihtiyacına göre girdi ve çıktı dosya formatlarının geliştirilmesine dair ipuçları verdiği için faydalı olmuştur. Google dosya sistemi üzerinde ne tür yazılımlar için MapReduce yönteminin kullanıldığına dair aydınlatıcı bir çalışmadır. Literatürde daha sonra DFS'lerde en yaygın olarak kullanılan dağıtık veri işleme yöntemi olması dolayısıyla en sık referans verilen çalışmalardan biri olmuştur.

Bo Dong ve arkadaşları [8] HDFS üzerinde çok büyük miktarda tutulan çoklu-medya verisinin saklanması ve işlenmesi hızını artırabilmek için geliştirdikleri yöntemi açıklamışlardır. Verilerin HDFS'ye kaydedilmeden önce önizleme (preview) verileri oluşturulup bu verilerin indeks yardımıyla birleştirilmesi ve daha sonra saklanması tekniğini geliştirmiş ve bu tekniği açıklamışlardır. Kullandıkları yöntem özetle HDFS'ye her PowerPoint dosyası kaydedilmek istendiğinde her bir sunu sayfası bir önizleme resim olacak şekilde küçük resimlere dönüştürülür, sonra bu resimler bir

indeks ile tek bir dosyada orijinal sunuyu da içerecek şekilde tekrardan birleştirilir ve HDFS'ye kaydedilir. Daha sonra işlenmesi gerektiğinde önizleme verileri kullanılarak daha çabuk istenen sonuç elde edilir. Bu çalışmalarında bizim çalışmamızda olduğu gibi imge işleme şeklinde değil de sunu dosyalarını kaydetme ve yükleme performansını artırmaya yönelik çalışmışlardır. Burada ön izleme verisi ile beraber gerçek veriyi de tutmalarının sebebi erişim hızını artırmaktır. HDFS'de önemli bir problem olan küçük boyutlu dosyaları tutma sonucu oluşan yavaşlığa çözüm niteliğinde çalışmalardan biridir.

Xuhui Liu ve arkadaşları [9] internet üzerinde GIS görüntülerini kaydeden ve kullanıcılara bu görüntüler üzerinde arama yapma imkânı veren Hadoop sunucularında küçük görüntü dosyalarının kaydından kaynaklı yavaşlığı gidermeye yönelik olarak dosyaların birleştirilerek HDFS'ye kaydını anlatan bir tekniği açıklamışlardır. Geliştirdikleri bir orta katman yazılımıdır, bu yazılım dosyaları kaydederken aynı düğümde olanları ve özellik olarak benzerlik taşıyanları indeksleyerek birleştirme yoluna gider. Sistemin aynı anda birçok kullanıcı tarafından kullanılırken, kullanıcıların genellikle izledikler GIS görüntülerinin koordinat olarak birbirini takip eden yakın yerlere ait olduğu bilgisini de dosyaları birleştirme algoritmalarına katmışlardır. Kullanılan teknik bu bilgiden yola çıkarak koordinat olarak birbirini takip eden alanlara ait GIS imgelerini birleştirerek guruplar. Böylelikle hem NameNode tarafından kullanılan hafıza miktarını azaltırken, hem de kullanıcıların dosyalara erişim hızını artırmışlardır. Bizim çalışmamızdan farklı olarak imgelerin kaydedilme ve erişilme performansını artırmaya yönelik bir çalışmadır. Bizim çalışmamızda ise imgelerin en hızlı şekilde işlenmesinin sağlanmasına yönelik bir eklenti geliştirilmesi sağlanmıştır. Liu ve arkadaşlarının bu çalışmaları, sadece HDFS için değil, aynı zamanda PVFS [10], pNFS [11] ve GlusterFS [12] gibi diğer bazı geleneksel DFS'ler için de uygulanabilir olması yönüyle bu alanda çok sık referans verilen bir makale olmuştur.

M. Vaidya [13] HDFS ve MapReduce modelini anlattığı çalışmasında, öncelikle GFS'te MapReduce yönteminin nasıl uygulandığını anlatmıştır. Bu çalışmasında HDFS'nin MapReduce görevlerini nasıl çalıştırdığını dosya sistemi ve işleyiciler kapsamında detaylı olarak açıklamıştır. Map ve Reduce işleyiciler arasında girdi ve çıktı yöntemlerini grafiksel olarak anlatmıştır. Ortanca veri (intermediate data)

oluşumu ve kullanımını açıklamıştır. Yönetici ve işçi düğümler arasında veri akışının nasıl olduğunu ve HDFS'de veri yerelliği (Data Locality) prensibinin nasıl uygulandığını açıklamaktadır. Bu tez kapsamında MapReduce hakkında detaylı bilgi edinimi amacıyla kullanılmıştır.

K. Wiley ve arkadaşları [14] önümüzdeki on yıl içerisinde uydulardan ve teleskoplardan elde edilen günlük uzay görüntüsü boyutlarının terabyte seviyesine ulaşacağını ve bu kadar verinin kaydının ve paralel işlenmesinin gerekliliğinden bahsetmişlerdir. Kendi geliştirdikleri ve Hadoop üzerine kurulu sistemde kayıtlı teleskop görüntülerini MapReduce yöntemiyle nasıl önişlemeye tabi tuttuklarını açıklamışlardır. Bu çalışmalarında veritabanında belli uzay alanına ait görüntü kümeleri içerisinde sorgulanarak elde edilen imgeler, işleyiciye girdi parçacığı olarak verilmiştir. Map fonksiyonları içerisinde örneğin aynı alanı içeren görüntüler, arka-fon çıkartımı (background subtraction), özellik çıkartımı (feature extraction) ve çakıştırma (warping) yöntemleriyle detay düzeyi ve netliği artırılarak daha detaylı bir sonuç görüntüsü elde edilmektedir. Bizim çalışmamızda olduğu gibi c++ dilinde yazılmış kendi görüntü işleme kütüphanelerini Hadoop içerisine JNI ile entegre etmişlerdir. Hadoop'un farklı kullanım alanlarını örneklemek açısından da önemli bir çalışmadır.

X. Qiu ve arkadaşları [15] kullandıkları biyoinformatik yazılımlarını ve algoritmalarını Hadoop üzerinde MapReduce yöntemiyle çalıştırarak performansını sergilemişlerdir. Veri yoğun çalışan yazılımlarında Hadoop'un performansı artırdığı görülmüştür. Bu çalışma kapsamında Hadoop'a ek olarak Dryad [16] gibi başka paralel veri işleme sistemleri de kullanılarak Hadoop ile performans karşılaştırılması yapılmıştır. Bulut hesaplama (cloud computing) alanında kullanılan en son teknolojilerden uygulama kapsamından bahsetmesi açısından faydalı bir çalışma olmuştur. Ayrıca Hadoop'un kullanılabileceği farklı uygulamaları göstermesi açısından da önemlidir.

M. Krishna ve arkadaşları [17] internet üzerinden web sayfası ve içeriği kaydı (web crawling) yaparken, oluşan çok büyük boyutlu verinin işlenmesi için Hadoop teknolojisini nasıl kullandıklarını açıklamışlardır. Bu çalışmada indirilen web sayfaları içeriklerinde bulunan çoklu-medya dosyalarının sakıncalı bir takım

görüntüler içerip içermediği ve buna göre sınıflandırılması amaçlanmıştır. Bunun için indirilen web sayfalarındaki çoklu-medya dosyaları HDFS'ye aktarılırken bulundukları web sayfaları ve konumlarına ait bilgiler de metadata bilgisi olarak kaydedilmiştir. Çoklu-medya verisi daha sonra MapReduce modeline göre hazırlanmış görüntü işleme görevleri ile çalıştırılarak içerik kontrol ve sınıflandırılmasının nasıl yapıldığı açıklanmıştır. Yapılan çalışma kapsamında kullanılan dosya formatları ve işleme tekniği detayları açıklanmamıştır fakat genel olarak kullandıkları mimariden bahsetmişlerdir. İlerde büyük boyutlu web sayfası verisini HDFS'de depolama ve işleme gerektiren bir sistem kurmak gerekirse bu makaledeki tasarımdan faydalanılabilecektir.

3. TEMEL ANLATIMLAR

Son yıllarda geliştirilen teknolojiler takip edildiğinde tüm işlemlerin bilgisayar tabanlı yapıldığı, bu nedenler bilgisayar kayıt kapasitelerini ve hızı artırmaya yönelik çalışmaların revaçta olduğu görülecektir. Artırılan kapasitelerle birlikte toplanan ve saklanan bilgilerin işlenmesi problemi de aşılmaya çalışılmaktadır.

Yaygın internet kullanımı özellikle saklanan verinin çok hızlı bir şekilde artmasına neden olmuştur. Her gün binlerce kişisel video, resim ve diğer çoklu-medya öğeleri internetteki kişisel sayfalara yüklenmekte veya sosyal paylaşım sitelerinden paylaşılmaktadır. Bu durum da sunucularda tutulan verinin hiçbir zaman olmadığı kadar artmasına neden olmuştur. Bu verilere ek olarak kurumsal verilerin çeşitliliği ve detayı da arttığından bu verilerin de boyutlarında ciddi bir artış olmuştur. Böylelikle petabyte boyutlarında verilerin tutulması ve işlenmesi günümüzde çok gerekli olmuştur.

Çok büyük boyutta kayıtlı verinin işlenmesi için öncelikle bu kayıtların erişim ve yükleme hızını artırıcı algoritmalar geliştirilmiştir; örneğin Google tarafında geliştirilen BigTable [18] algoritması ve Facebook tarafından geliştirilen Cassandra algoritması [19] ile bu tür çok büyük boyutlu kaydedilmiş veriye erişimi hızlandırmayı amaçlamışlardır. Başka pek çok firma da bu alanda performansı artırabilmek için ciddi yatırım yapmışlardır.

Google tarafından tanıtılan MapReduce [3] programlama modeli de bunlardan biridir. MapReduce ile dağıtık olarak kaydedilmiş yüksek miktarda veriyi ölçeklenebilir bir şekilde paralel olarak işlemek mümkündür. MapReduce büyük veriyi paralel işlemek için verinin parçalanmasını ve işleyicilerin buna göre oluşturulmasını kullanıcıdan gizleyerek arka planda gerçekleştirir, kendi içinde hata toleransı yönetimi de vardır. Bu da sistemin kullanıcısının kendi problemine odaklanmasını ve dağıtık sistemin yönetimiyle ilgilenmemesini sağlar.

MapReduce yöntemi yaygın olarak web sayfalarının kaydedilmesiyle (web crawling) elde edilen büyük veri kümelerini işler. Bunların içinde kelime veya diğer aramalar yapar. Bunun dışında son dönemde özellikle büyük boyutlu kurumsal veriler üzerinde veri madenciliği yapma amaçlı da kullanılmaya başlanmıştır [20]. Astronomik görüntülerin işlenmesi [14] veya biyoinformatik uygulamaların [15] paralel çalıştırılması gibi belli alanlarda özelleştirilerek de kullanılmaktadır. Dolayısıyla MapReduce, çok büyük boyutlu kaydedilmiş farklı verilerin hızlı bir şekilde paralel olarak işlenmesine yönelik tasarlanmıştır. Buna Google arama motoru ile yapılan aramaların hızlı bir şekilde işlenip sonuçlandırılmasını örnek olarak verebiliriz. Bu normal bir veritabanı sistemi ile yapılamayacak şekilde paralelleşmiş ve hızlanmış bir çalışma şeklidir. Veritabanı yönetim sistemleri (DBMS) tekil veriyi daha hızlı sorgulayabilir [21] fakat paralel işlenmesi gereken milyonlarca veri söz konusu olduğunda MapReduce en hızlı çözüm olmaktadır. MapReduce bu paralelleştirmeyi yaparken yük dağılımı, hata yönetimi ve paralelleştirmenin detaylarını kullanıcıya yansıtmaz.

Hadoop Apache Software Foundation tarafından MapReduce programlama modeli üzerinde java diliyle geliştirilmiş bir dağıtık dosya sistemi ve paralel işleme altyapısıdır. Yaygın olarak kullanılan metin işleme kütüphanesi olan Apache Lucene'in de [22] geliştiricisi olan Doug Cutting tarafından geliştirilmiştir. Lucene projesinin bir parçası olarak 2002 de öncelikle Apache Nutch [23] açık kaynak kodlu web arama motoru geliştirildi. Fakat bu arama motoru tüm dünyadaki web sayfalarını indeksleyip arama yapacak kapasitede değildi. 2003 yılında yayınlanan Google Dosya Sistemi'ni [2] açıklayan makaleden de etkilenerek Nutch Dağıtık Dosya Sistemi'ni (Nutch Distributed File System (NDFS)) geliştirdiler. 2004 yılında MapReduce hakkında yapılan çalışmaları da sistemlerine uyarlayarak, 2008'in Ocak ayında Hadoop Dağıtık Dosya Sistemi'ni (HDFS) ortaya çıkardılar. HDFS çok büyük miktarda kayıtlı veriyi dağıtık olarak kaydetmeyi ve paralel olarak MapReduce modeliyle işlemeyi sağlayan bir sistem olarak yaygın bir şekilde kullanılmaktadır.

3.1. MapReduce Programlama Modeli

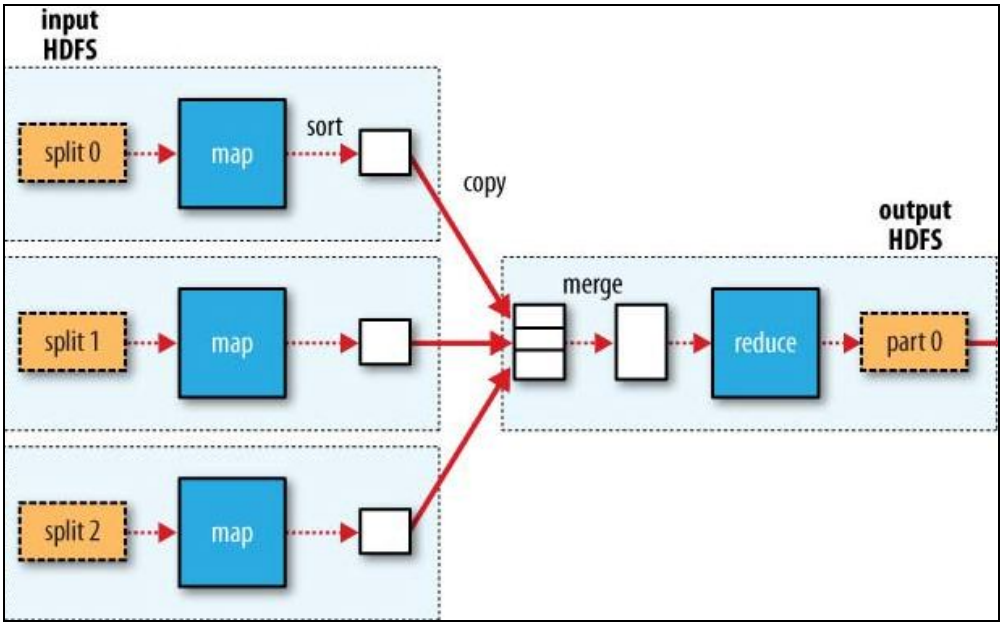
MapReduce [3] büyük veri kümelerindeki verilere en hızlı şekilde erişip işlemeyi sağlayan bir programlama modelidir. İlk olarak Jeffrey Dean ve Sanjay Ghemawat

tarafından Google sunucularında kayıtlı verilerin paralel işlenmesi amacıyla geliştirilmiştir. Günlük olarak kaydedilen web sayfalarının kütüğe yazılması sonucu oluşan veri kümesi içerisinde çok kısa sürede sonuç üretecek şekilde arama yapmak için geliştirilmiştir. Binlerce paralel çalışan bilgisayarı kullanabilecek şekilde tasarlanmıştır. Bu noktada dağıtık sistemlerdeki verinin parçalanması, dağıtılması ve işlemin paralelleştirilmesi gibi problemler ortaya çıkmıştır. Bu problemleri aşabilmek için yük dengeleme (load balancing), verinin dağıtılması ve hata toleransı işlemleri için bir soyutlama modeli tasarladırlar. Geliştirdikleri MapReduce yöntemi bu nedenlerle kullanıcı tarafından kolayca kullanılabilir ve dağıtık sistem üzerinde kolayca işlem yapmaya izin veren bir arayüze sahiptir. Basit ve güçlü olan bu sistem sayesinde kullanıcı sadece map ve reduce fonksiyonlarını yazarak paralelleştirilmesini istediği işi tanımlar, sistem bu iş için gerekli veri bölütlenmesini, paralelleştirmeyi ve hata yönetimini kendi yönetir. MapReduce kullanarak paralelleştirilebilen farklı uygulamalar mevcuttur. Bunlara örnek olarak web sayfası erişim frekansı tarama ve bulma [23], arama motorlarında kelime veya dosya arama [2], Büyük boyutlu astronomik imgeler üzerinde işlem yapmak [14], Dağıtık olarak sıralama algoritmaları çalıştırmak ve Paralel görüntü işlemek [17] gibi paralel hesaplama gerektiren bazı işlemleri verebiliriz.

MapReduce modeli daha sonra Apache yazılım topluluğu tarafından Hadoop adında açık kaynak kodlu bir proje olarak yeniden geliştirilmiştir. Hadoop ücretsiz bir sistem olduğundan çok geniş bir geliştirici topluluğu tarafından kullanılmaktadır. Hadoop kullanıcıları arasında Yahoo gibi büyük kurumlar da mevcuttur.

Hadoop'ta paralelleştirilecek hesaplama bir görev (job) olarak tanımlanır. Bu görev geliştirici tarafından kodlanan map ve reduce fonksiyonlarını içerir. Bu soyutlamada ilham kaynağı Lisp gibi fonksiyonel dillerdeki benzer fonksiyonlar olmuştur. MapReduce sistemi bu fonksiyonların birçok kopyasını paralel olarak farklı makinelerdeki işleyicilerde çalıştırır. Map fonksiyonu anahtar/kilit (key/value) ikilisinden oluşan bir kayıt bilgisini girdi olarak alır ve yine anahtar/kilit ikilisinden oluşan bir çıktı verisini oluşturur. Çok sayıda paralel çalışan bu Map fonksiyonları Hadoop sisteminde çıktılarını ortanca çıktı (intermediate output) olarak tutarlar ve bu çıktılar reduce fonksiyonlarına girdi olarak gönderilir. Tanımlı reduce fonksiyonlarını çalıştıran işleyiciler bu ortanca çıktıları girdi olarak kabul edip Şekil

3.1.'deki gibi bunları tekrardan düzenleyip birleştirerek nihai çıktıyı oluşturur. Bu yeniden düzenleme sırasında aynı anahtar değerine sahip olan ortanca çıktıların değerleri birleştirilir.



Şekil 3.1. MapReduce veri akış diyagramı [4]

Hadoop sistemi tanımlanmış olan bu fonksiyonları otomatik olarak paralelleştirerek büyük bilgisayar kümeleri üzerinde dağıtık olarak çalıştırır. Daha önce de bahsedildiği gibi verinin dağıtılması, işlerin sıralanması, yönetilmesi, dosya sisteminden işleyicilere verilerin iletilmesi, makineler arası haberleşme ve hata yönetimi gibi işler sistem tarafından otomatik olarak çözümlenir.

3.1.1. MapReduce çalışma şekli

MapReduce çalışma şeklini biraz daha detaylı incelemeden önce bu tez içerisinde kullanılan bazı kavramları açıklamamız gerekmektedir.

Makine: Dağıtık sistemi oluşturan fiziksel bilgisayarlardan her birine denir.

İşleyici (Task): Her makine de çalışan paralelleşmiş her bir MapReduce iş parçacığı. Map işleyicisi ve Reduce işleyicisi olarak tanımlanır.

Düğüm (Node): Hadoop tarafından her bir makineye verilen dağıtık sistemdeki rol veya tanım. Her makine aslında bir düğüm olarak görev almaktadır.

Girdi Parçacığı (Input Split): İşleyicilerin dosya sisteminden okudukları ve işleyecekleri veri parçacığıdır.

Ortanca Çıktı (Intermadiate Output): Map fonksiyonları tarafında oluşturulan ara çıktı verisi. Bu veri anahtar/değer ikilisi şeklinde olup Reduce fonksiyonuna girdi olarak verilir.

MapReduce görevinin çalışma şeklini aşağıdaki adımlar şeklinde inceleyebiliriz:

a. Hadoop MapReduce sisteminde kayıtlı olan büyük boyutlu dosyalar öncelikle dosya sistemi varsayılan blok boyutu olan 64 megabyte boyutundaki parçacıklara ayrılarak bu parçacıklar farklı düğümlerde kaydedilir. Bir parçanın birden fazla kopyası sistemde olabilir (Replication), bunun sebebi hata oluşan düğümdeki verinin kaybolmasını engellemektir. Ayrıca replication yapılan veri paralel işleme sırasında daha fazla düğümde bulunacağından veri yerelliği daha fazla elde edilmiş olur, bu durum da daha az ağ üzerinden veri transferine ve daha hızlı çalışmasını sağlar.

b. HDFS sistemindeki düğümlerden bir tanesi yönetici (master) düğümdür. Yönetici düğüm diğer işçi düğümlere işleyici olarak görevler verir. Bir işleyici Map işleyicisi veya Reduce işleyicisi olarak farklı görev alabilir. Sistemdeki Map işleyicisi sayısı işlenecek veri boyutlarına göre otomatik olarak belirlenir. Reduce sayısı kullanıcı tarafından görev ayarları yapılırken belirlenebilir.

c. Map işleyicisi girdi parçacığı verisini sistemden okuyarak bundan map fonksiyonu için anahtar/değer ikilisini içeren kayıt verilerini oluşturur. Kayıt girdileri ile kullanıcı tarafından oluşturulmuş olan map fonksiyonları çağrılır. Map fonksiyonlarının oluşturduğu ortanca çıktı ilgili makine tarafından reduce fonksiyonuna gönderilmek üzere geçici hafızaya ve sonra da yerel diske kaydedilir. Reduce fonksiyonuna gönderildikten sonra bu veri silinir.

d. Oluşturulan ortanca çıktı verileri Reduce işleyicilerine paylaştırılır ve Reduce işleyicileri bu verileri okuyarak ortanca anahtar verilerine göre sıralar ve guruplar. Böylelikle aynı anahtar değerine sahip kayıtların değerleri

gruplandırılmış olur. Kullanıcı isterse bu gruplama ve sıralama şeklini değiştirebilir.

e. Reduce fonksiyonu içerisine gruplandırılmış yeni anahtar/değer ikilileri girdi olarak verilir. Kullanıcı tarafından oluşturulmuş reduce fonksiyonu nihai işleme tabi tutup oluşturduğu çıktıları çıktı dosyasına ekler.

f. Yönetici düğüm, görevi çalıştıran kullanıcıya görevin tamamlanma bilgilerini sunar ve görevi sonlandırır.

3.1.1.1. Map fonksiyonu

Hadoop'ta map fonksiyonu tanımlayabilmek için Mapper arayüz sınıfını işletmek gerekmektedir. Aşağıdaki map fonksiyonu örneğinde görüldüğü gibi MapReduceBase sınıfından türeyen ve Mapper arayüz sınıfını işleten bir sınıf yazılır. Burada dört tane parametre ile işletilen Mapper arayüzü için sırasıyla girdi anahtar formatı, girdi değer formatı, çıktı anahtar formatı ve çıktı değer formatı şeklindedir. Örnekte bunlar "<LongWritable, Text, Text, IntWritable>" şeklinde tanımlanmıştır. Bu parametre sınıfları Hadoop içinde tanımlanmış sınıflardır ve org.apache.hadoop.io paketinden diğer formatlara da ulaşılabilir.

Map fonksiyonunun tanımına baktığımızda girdi anahtar değer parametreleri formatları ile birlikte OutputCollector ve Reporter objelerini de parametre olarak aldığını görürüz. Burada OutputCollector sınıfı Map fonksiyonunun çıktısını oluşturmak için kullanılır ve çıktı bu sınıfın "collect" fonksiyonuna girdi olarak verilir. Reporter sınıfı kullanılarak da map fonksiyonunun ilerlemesi ile ilgili bilgiler veya çıkan hata ve uyarılar yönetici düğüme gönderilir. Şekil 3.2.'de sergilenen örnek program kodundabüyük boyutlu meteoroloji verisinden yıla bağlı sıcaklık listeleri taranarak paralel olarak çıkartılmaktadır. Burada meteroloji verisini tutan sunucuda yıllar boyunca kaydedilmiş olan kütükleme verisi içerisindeki her bir satır,map fonksiyonuna işlenmek üzere aktarılır. Map fonksiyonu gelen satır bilgisini metin formatına dönüştürür. Daha sonra bu String verisi içerisinde yıl bilgisini içeren karakterlerin karşılık geldiği 15-19 aralığı içerisindeki karakterleri okuyarak burdan yıl bilgisini elde eder. 88-92 indeksli karakterlerden de sıcaklık bilgisini elde eder ve

bu bilgileri <anahtar, değer> ikilisi şeklinde collect medodunu çağırarak çıktı olarak oluşturur.

```
public class MaxTemperatureMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output, Reporter reporter)
                    throws IOException {
                    String line = value.toString();
                    String year = line.substring(15, 19);
                    int temperature;
                    if (line.charAt(87) == '+') {
                            temperature = Integer.parseInt(line.substring(88, 92));
                    } else {
                            temperature = Integer.parseInt(line.substring(87, 92));
                    }
                    output.collect(new Text(year), newIntWritable(temperature));
}
```

Şekil 3.2. Örnek Map fonksiyonu program kodu

3.1.1.2. Reduce fonksiyonu

Reduce fonksiyonu, MapReduceBase sınıfından türeyen ve Reducer arayüzünü işleten bir Reducer sınıfı içerisinde yer alır. Map işleyicinin oluşturduğu ortanca çıktı kayıtları gruplanarak reduce fonksiyonuna verilir. Reduce fonksiyonu gelen kayıtları en son işleme tabi tutarak OutputCollector sınıfına ait "collect" metodu ile çıktı olarak HDFS'ye kaydeder. Şekil 3.3.'te sunulan örnek reduce fonksiyonu kodunda map fonksiyonundan alınan çıktı kayıtlarındaki yıla bağlı en yüksek sıcaklık değerleri bulunarak çıktı dosyasına yazılmıştır. Bulunan en yüksek sıcaklık değerleri "value" olacak ve yıl bilgisi de "key" olacak şekilde elde edilen kayıt verileri OutputCollector sınıfı yardımıyla çıktı olarak yazılır. Burada Reporter nesnesi gerektiğinde ilgili fonksiyonun durumunu JobTracker'a iletmek için kullanılabilmektedir. Normalde map fonksiyonu çıktısı ile oluşan ortanca çıktı (intermediate output) olup tüm görevin nihai çıktısı değildir. Reducer'lardan elde edilen çıktı ise direkt olarak HDFS'ye kaydedilir ve nihai çıktıdır. Hadoop bu çıktı

dosyasına kendi bir isim verir.(örn: part-0001) Bu isimlendirmede görevin ve reducer'ın tanımlama numaraları kullanılır.

```
public class MaxTemperatureReducer extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
                    OutputCollector<Text, IntWritable> output, Reporter
                    reporter)
                    throws IOException {
                    int maxValue = Integer.MIN_VALUE;
                    while (values.hasNext()) {
                        maxValue = Math.max(maxValue, values.next().get());
                    }
                    output.collect(key, new IntWritable(maxValue));
    }
}
```

Şekil 3.3. Örnek Reduce fonksiyonu program kodu

3.1.1.3. MapReduce görevi tanımlama

Map ve reduce fonksiyonlarını içeren Mapper ve Reducer sınıflarını tanımladıktan sonra işleyicileri oluşturmak için öncelikle bir Hadoop görevi tanımlamamız gerekmektedir. Şekil 3.4.'teki örnek kod parçacığında görülebileceği gibi JobConf sınıfından oluşturulan nesne, göreve ait özelliklerin atandığı ve görevin nasıl çalıştırılması gerektiğinin ayarlarının yapılabileceği nesnedir. Hadoop'ta görev çalıştırıldığı zaman görevi tanımladığımız bu sınıf ve Mapper ve Reducer sınıfları ile birlikte bir java çalıştırılabilir dosyasına (Java Archive (JAR)) dönüştürülür. Bu dosya dağıtık sistemde görevi çalıştıracak tüm düğümlere gönderilir. JobConf sınıfı bu görevin düğümlerde nasıl çalıştırılacağı bilgilerini içerdiği için çok önemlidir.

Bu tanımlama yapılırken öncelikle girdi ve çıktı dosyalarının nereden okunacağı ve nereye yazılacağına dair adres bilgileri FileInputFormat ve FileOutputFormat sınıfları içinde tanımlanır. "AddInputPath" metodu kullanılarak birden fazla girdi klasörü veya dosyası da görev için tanımlanabilir. Çıktı dosyasının adresi tanımlandıktan sonra görev çalıştırıldığında eğer HDFS'de bu isimde bir dosya

mevcutsa görev hata vererek çalışmaz. Bu klasörün sistemde olmaması görev çalıştığında görevin bu klasörü yaratması açısından gereklidir.

"SetOutputKeyClass" ve "SetOutputValueClass" fonksiyonları ile çıktı dosyasına yazılan kayıtların anahtar ve değer ikililerinin formatları belirlenir. Tüm yapılandırma bittikten sonra "runJob" fonksiyonu çağrılarak görev çalıştırılır ve görevin sonlanması beklenir.

```
public class MaxTemperature {
    public static void main(String[] args) throws IOException {
        JobConf conf = new JobConf(MaxTemperature.class);
        conf.setJobName("Max temperature");
        FileInputFormat.addInputPath(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        conf.setMapperClass(MaxTemperatureMapper.class);
        conf.setReducerClass(MaxTemperatureReducer.class);
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        JobClient.runJob(conf);}
}
```

Şekil 3.4. Örnek MapReduce görevi tanımlama program kodu

3.1.1.4. MapReduce görevi çalıştırma

Kullanıcı Hadoop görevini çalıştırmak istediğinde komut satırından "hadoop" komutunu ardından JAR dosyasının adı ile beraber çağırır. Aşağıda komut satırından çalıştırma ve çalışma tamamlanınca Hadoop'un komut satırına yazdırdığı mesajlar görülebilir. Burada zaman bilgileri ile birlikte görevin sonlanması sonucunda map ve reduce işleyicilerinin istatistikleri sergilenmektedir.

```
% hadoop MaxTemperature input/ncdc/sample.txt output

09/04/07 12:34:36 INFO mapred.JobClient: map 100% reduce 100%

09/04/07 12:34:36 INFO mapred.JobClient: Job complete: job_local_0001

09/04/07 12:34:36 INFO mapred.JobClient: Counters: 13
```

09/04/07 12:34:36 INFO mapred.JobClient: FileSystemCounters

09/04/07 12:34:36 INFO mapred.JobClient: FILE_BYTES_READ=27571

09/04/07 12:34:36 INFO mapred.JobClient: FILE_BYTES_WRITTEN=53907

09/04/07 12:34:36 INFO mapred.JobClient: Map-Reduce Framework

09/04/07 12:34:36 INFO mapred.JobClient: Reduce input groups=2

09/04/07 12:34:36 INFO mapred.JobClient: Combine output records=0

09/04/07 12:34:36 INFO mapred.JobClient: Map input records=5

09/04/07 12:34:36 INFO mapred.JobClient: Reduce shuffle bytes=0

09/04/07 12:34:36 INFO mapred.JobClient: Reduce output records=2

09/04/07 12:34:36 INFO mapred.JobClient: Spilled Records=10

09/04/07 12:34:36 INFO mapred.JobClient: Map output bytes=45

09/04/07 12:34:36 INFO mapred.JobClient: Map input bytes=529

09/04/07 12:34:36 INFO mapred.JobClient: Combine input records=0

09/04/07 12:34:36 INFO mapred.JobClient: Map output records=5

09/04/07 12:34:36 INFO mapred.JobClient: Reduce input records=5

Bu mesajlardan Hadoop'un görevi çalıştırma sırasında göreve verdiği kimlik bilgileri görülebilir, örneğin aşağıdaki çalıştırma için göreve job_local_0001 etiketini vermiştir. Çalıştırma sonucunda "output" klasöründe oluşan dosyanın adı “part-0000” dosyasıdır. Bu dosyanın içeriğini her hangi bir metin editör programı ile incelediğimizde yukarıdaki Map ve Reduce fonksiyonları için çalışan görevin çıktısının her satırında bir yıl bilgisi, karşısında da o yıla ait en yüksek sıcaklık bilgisinin olduğunu görürüz.

3.2. HDFS (Hadoop Dağıtık Dosya Sistemi)

HDFS çok büyük miktarda veya büyük boyutta dosyaların dağıtık bir sistemde saklanması ve bu dosyalara en hızlı ve güvenli bir şekilde erişilebilmesi için tasarlanmıştır [5]. MapReduce prensiplerine uygun olarak verilerin kolay işlenebilmesine uygun bir mimari kullanılmıştır.

Normal dosya sistemlerinde tutulabilen boyutların çok üzerinde, petabyte boyutlarında dosyaların saklanıp işlenebilmesi için; HDFS saklanacak dosyaları blok boyutunda (örn: 64 MB) parçalara ayırır ve parçaları farklı düğümlerde saklar. Böylelikle binlerce düğüm üzerinde petabyte boyutunda dosyaların saklanması mümkün olabilmektedir.

Dosyaların parçalar halinde dağıtık olarak saklanması, bu dosyalar üzerinde çalışacak olan işleyicilerinde kendi bulundukları düğümdeki dosyaya ait parçayı işlemesine imkân verdiğinden, bu şekilde çalışması MapReduce işleyicilerinin de veriye daha çabuk ulaşarak daha hızlı işlem yapmasını sağlamıştır. Veri yerelliği (Data Locality) [5] prensibi denen bu yapı HDFS'nin paralel işleme performansını da ciddi anlamda artırmaktadır. Veri yerelliği prensibinde, işlenecek verinin sürekli olarak işleyici olan düğümlere ağ üzerinden transferi gerekmemektedir, dolayısıyla ağ gecikmesinden kaynaklanan yavaşlamaları azaltmaktadır.

3.2.1. HDFS NameNode ve DataNode yazılımları

HDFS dosya sistemi iki tür düğümden oluşur bunlar: dosya sistemi yönetici düğümünde bulunan NameNode ve işçi düğümlerde bulunan DataNode yazılımlarıdır. NameNode yazılımı, dosya sisteminde kayıtlı bulunan tüm dosya ve klasörlere ait dosya sistem ağacını tutar ve yönetir. Her dosyaya ait kayıt bilgisi (Metadata) NameNode tarafından yerel diskte tutulur. NameNode ayrıca bir dosyaya ait blokların bazılarının tutulduğu DataNode'lara ait bilgileri de tutar. Fakat yerel blok adreslerini yerel olarak tutmaz sadece hangi bloğun hangi DataNode tarafından yönetildiği bilgisini tutar.

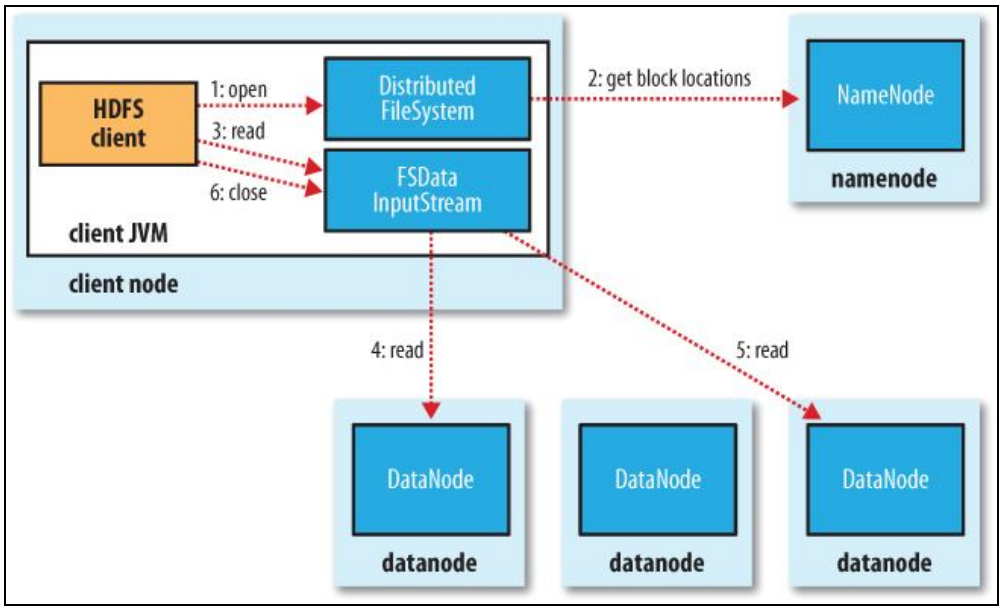
DataNode ise kendinde bulunan blokların saklanması eklenmesi silinmesi ve adreslenmesi gibi işleri kontrol eder. Dolayısıyla DataNode yazılımları işçi

düğümlerde bulunan blok işlemlerinin yürütüldüğü ve yapılan işlemleri ve blok listelerini NameNode yazılımına raporlayan bir yazılımdır.

NameNode yazılımı tüm dosya sistemini yönettiğinden bozulması durumunda tüm dosya sistemi ve dosyalar kaybolacaktır. Hadoop bunun önüne geçmek için iki mekanizma geliştirmiştir. Bunlardan ilki: NameNode'un tüm dosya sistemi ağaç verisini belli aralıklarla imajını almasıdır, ikincisi ise paralelde çalışan ikincil bir NameNode çalıştırmak mümkündür. Bu ikincil (Secondary) NameNode bilgilerini yedekleme amaçlı çalışır ve genellikle ayrı bir fiziksel makinede bulunur.

3.2.2. HDFS girdi çıktı mekanizması (HDFS I/O)

Kullanıcı bir dosyayı HDFS'de okumak istediği zaman bununla ilgili istekte bulunduktan sonra HDFS'nin bunu gerçekleştirme mekanizması aşağıda Şekil 3.5.'te sergilenmiştir.

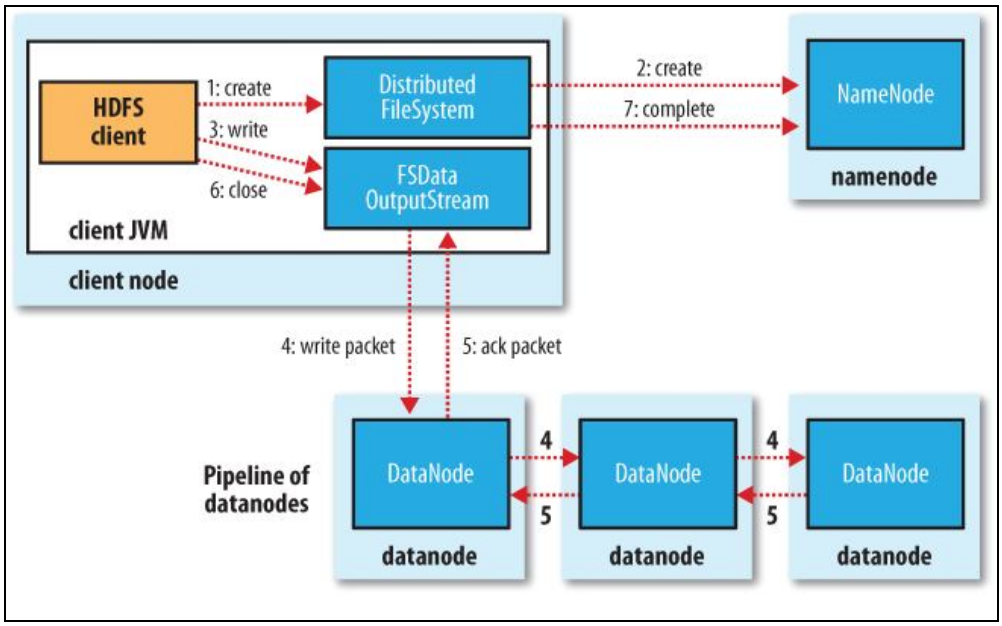


Şekil 3.5. HDFS'den istemci tarafından dosya okunma şeması [4]

Kullanıcı HDFS'de open() ile DistributedFileSystem nesnesinin NameNode ile haberleşmesini başlatır. Bunu gerçekleştirmek için Remote Procedure Call (RPC) [24] mekanizmasını kullanır. Böylelikle istenilen dosyaya ait blokların bulunduğu DataNode'lara ait adresleri NameNode geri döndürür. Bu gelen DataNode'lardaki blokları adreslerinden okumak için FSDataInputStream üzerinden read() fonksiyonu çağrılır. Bunu her blok bilgisini okumak için tekrar yapar. Tüm blok bilgileri

okunduktan sonra da FSDataInputStream üzerinde close() fonksiyonu kullanarak veri akışı durdurulur. Eğer veriyi DataNode'lardan okuma sırasında problem oluşursa okunacak blok bilgisini içeren diğer node üzerinden bu blok tekrardan okunur. Eğer blok verisinin kontrol-toplam (checksum) bilgisi kontrol edilerek bozuk olduğu anlaşılırsa, bu durum NameNode'a bildirilir ve yedeklenmiş blok bilgisi (Replication) ile bozuk blok verisi güncellenir. NameNode her zaman istemciye en yakın DataNode'da blok mevcutsa onun adresini döner bu da erişim sürelerini kısaltan performansı artıran bir mekanizmadır.

Dosyaların kullanıcı tarafından HDFS'ye yazılması da aşağıda Şekil 3.6.'da sergilenmiştir. Burada izlenen ana sıra, önce dosyanın yaratılması, sonra verisinin dosyaya yazılması, ardından da dosyanın kapatılmasıdır.



Şekil 3.6. HDFS'ye istemci tarafından dosya yazılma şeması [4]

Kullanıcı tarafından dosya DistributedFileSystem nesnesi üzerinden create() fonksiyonu çağrılarak yaratılır. DistributedFileSystem NameNode'a bir RPC [24] mesajı ile dosya sisteminde yeni bir dosya yaratması komutunu gönderir. İlk oluşturulan bu dosya bilgisi hiç bir blok verisi içermez. NameNode eğer aynı isimde dosya yoksa istemcinin yazma haklarını kontrol ederek bu isteği gerçekleştirir. DistributedFileSystem dosya içeriğinin yazılması için istemciye bir FSDataOutputStream nesnesi oluşturur. Bu kendi içinde yazılacak olan veriyi paketlere parçalar. Bu parçalar daha sonra bloklar halinde sistemde bulunan

DataNode'lara kaydedilir. Bu blokların çoğaltılarak kopyalanması işlemi bir boru hattı (pipeline) mekanizmasına göre olur. İlk DataNode bloğu kaydeder ve ikinci DataNode'a gönderir o da kaydeder, daha sonra ikinci üçüncüye gönderir kaydedilmesi için. Tüm yazma işlemi tamamlandıktan sonra tekrardan FSDataOutputStream nesnesinin close() fonksiyonu çağrılarak kapatılır ve yazma sonlandırılmış olur.

3.2.3. HDFS'de MapReduce yöntemiyle paralel işleme

Kullanıcı ve geliştirici açısından Hadoop ile bir görevi çalıştırmak tek satırlık "JobClient.runJob(conf)" komutunu çalıştırmak kadar basit gibi görünse de kullanıcıdan gizlenen pek çok işlemin yönetilmesini gerektirmektedir. Aşağıda bir Hadoop görevini başlatıp, dağıtık sistem üzerinde çalıştırabilmek için gerekli olan bileşenler sıralanmıştır.

a. Öncelikle görevi tanımlayacak ve sisteme yükleyecek bir istemci olmalıdır.
b. JobTracker denen ana yönetici düğümde bulunan ve tüm görevin çalıştırılmasını yöneten yazılım çalışıyor olmalıdır.
c. İşçi düğümlerde TaskTracker denen ve iş parçacılarını çalıştıracak olan yazılımlar çalışıyor ve JobTracker ile haberleşiyor durumda olmalıdır.
d. Tüm bu yapı HDFS ile bağlantılı bir şekilde ilgili girdi çıktı verilerini kullanabiliyor durumda olmalıdır.

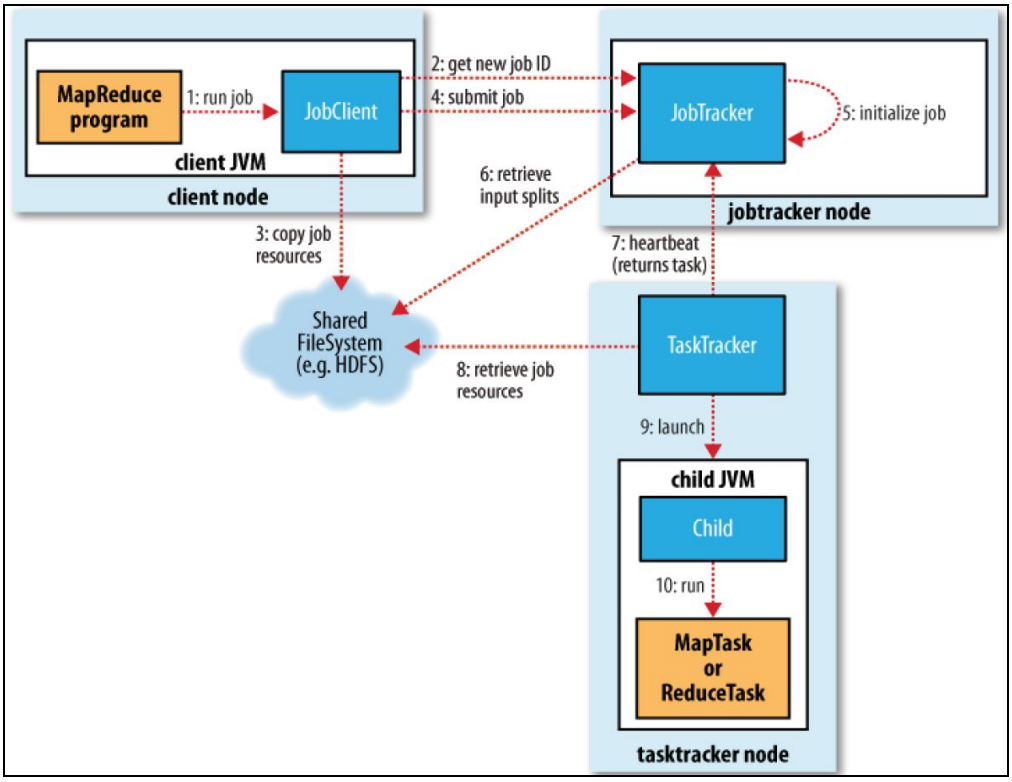
3.2.3.1. Hadoop paralel işleme mimarisi

Oluşturulan göreve ait yazılımda runJob() fonksiyonu çağrılarak yeni bir JobClient nesnesi oluşturulur ve bu nesne üzerinden Şekil 3.7.'de görüleceği gibi submitJob() fonksiyonu çağrılır. Görev yüklendikten sonra runJob() metodu içerisinde sürekli olarak görevin ilerlemesi kontrol edilir ve görev tamamlanınca ilgili göreve ait sayaç (counter) bilgileri kullanıcı konsoluna yansıtılır.

Buna göre bir hadoop görevinin çalışabilmesi için submitJob() ile görevin atanması sırasında aşağıdaki adımların gerçeklenmesi gerekmektedir:

a. JobTracker'dan yeni bir görev numarası oluşturması Şekil 3.7. adım 2'deki gibi istenir.

b. Görevin çıktı dosyasının durumu incelenir, eğer daha önceden bu isimde bir dosya mevcutsa veya çıktı dosyası tanımlanmamışsa görev sonlandırılarak hata mesajı yayınlanır.
c. Girdi parçacıkları görev için hesaplanır, eğer girdi dosyası bilgileri tanımlanmamışsa veya girdi dosyası parçalanamıyorsa hata mesajı yayınlanır.
d. Görevin bilgilerini ve çalıştırma fonksiyonlarını içeren JAR dosyası sistemdeki işçi düğümlerin erişimleri için sisteme dağıtılır.
e. JobTracker'a görevin çalıştırılmaya elverişli bir durumda hazır olduğu bildirilir.



Şekil 3.7. HDFS'de MapReduce görevinin çalıştırılma şeması [4]

JobTracker başlatılmaya elverişli olan görevi artık diğer görevlerle paralel çalıştırabilmek için kendi görev kuyruğuna yerleştirir. JobTracker'ın bir görevi de görevler arasında sıralama (scheduling) yapmaktır. Ardından görevi başlatmak için görevle ilgili alt iş parçacıklarının ön ayarlamasını yapar. Bunun için kendi içinde görevle ilgili bir nesne oluşturur, bu nesne görev çalıştırma bilgileri, oluşturulacak işleyicilerin listesi ve girdi parçacıklarının bilgisini içerir. Daha sonra JobTracker her bir girdi parçacığı için bir Map işleyicisi oluştur. Ayrıca Reduce işleyicilerini de

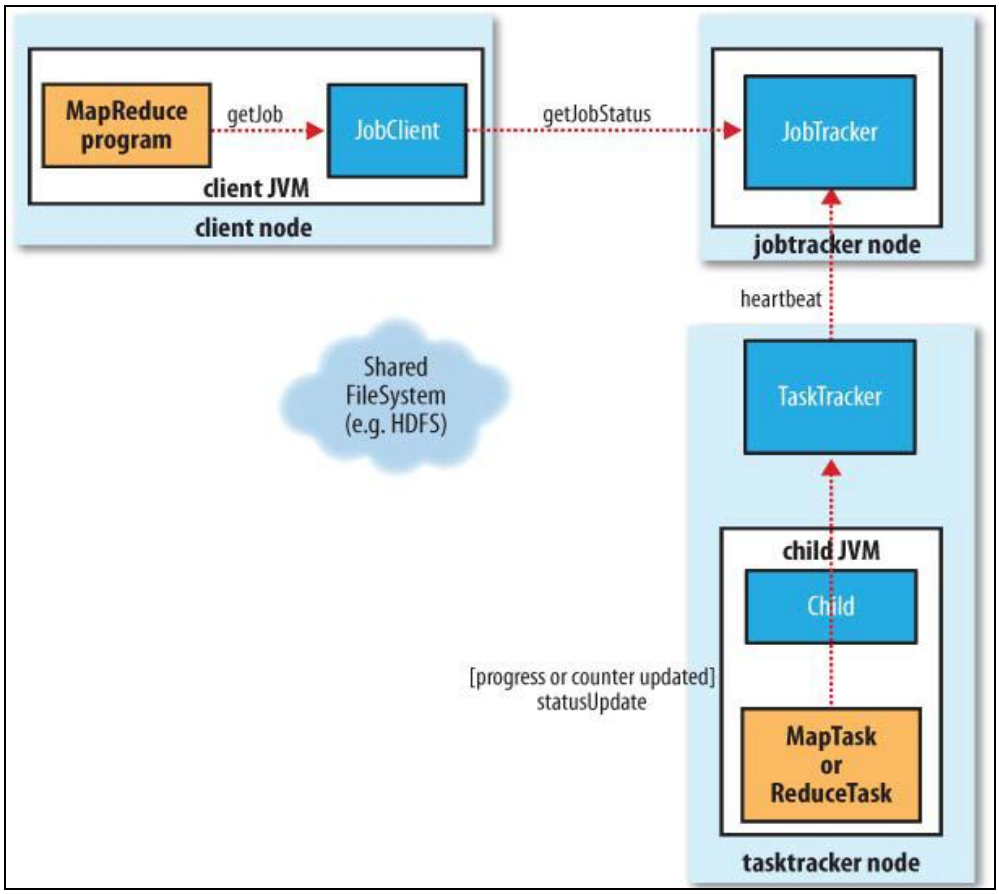
geliştirici tarafından setNumReduceTasks() metoduyla girilen değere göre oluşturur. Tüm bu işleyicileri yönetebilmek için her bir işleyiciye ayrı etiket numaraları verir.

TaskTrack'lar her bir işleyici düğümde bulunur ve sürekli olarak kalp atışı (heartbeat) şeklinde mesajlarla JobTracker'a kendi durumu hakkında bilgi verir. Bu mesaj kapsamında hem kendisinin halen ayakta çalışmakta olduğunu bildirir hem de yeni göreve hazır olma durumunu bildirir. JobTracker kullandığı sıralama algoritmasına göre sıradaki göreve ait bir iş parçacığını herhangi bir TaskTracker'a atayabilir. TaskTracker'ın bulunduğu bilgisayarın işlemci sayısına ve hafıza miktarına bağlı olarak bir TaskTracker aynı anda birkaç tane Map işleyicisi ve Reduce işleyicisini paralel olarak çalıştırabilir. Bir TaskTracker'a atanacak Map işleyicisi görevi belirlenirken veri yerelliği (Data Locality) [13] dikkate alınır. Böylelikle işleyici ile işleyeceği girdi parçacığının mümkün olduğunca yakın olması sağlanmış olur. Reduce işleyiciler için böyle bir seçim söz konusu değildir. Bu nedenle Map İşleyicilerin TaskTracker'lara atanma sıralamasında daha fazla önceliği vardır.

TaskTracker'a ilgili işleyici görevi atandıktan sonra TaskTracker'ın bu görevi çalıştırması gerekmektedir. Bunun için görev çalıştırma bilgilerini içeren çalıştırılabilir JAR dosyasını dosya sisteminden okur. Bunu yaparken ayrıca eğer çalıştırma için gerekli olabilecek başka dosyalar varsa bunları da Distributed Cache 'den [4] okur. İşleyici bir çalışma klasörünü kendi yerel dosya dizininde oluşturur ve JAR dosyasının içeriğini bu klasöre açar. Daha sonra TaskRunner oluşturarak görevi çalıştırır. TaskRunner her bir işleyiciyi çalıştırabilmek için ayrı bir java sanal makinası (Java Virtual Machine) çalıştırır, böylelikle map veya çalışan Reduce işleyicilerinde çıkan hatalar TaskTracker'ı etkilemez. İşleyici çalışırken Şekil 3.8.'de sergilendiği gibi sürekli olarak ilerlemesi hakkında bilgileri TaskTracker'a iletir. Bunun için kendi içinde çeşitli sayaçlar tutar. Tüm bu bilgiler JobTracker'a iletilir. Kullanıcı programı içerisinde isterse JobClient objesinden getJobStatus() fonksiyonuyla görevin herhangi bir andaki durum bilgisini JobTracker'dan alabilir.

JobTracker görevle ilgili çalışan en son işleyicinin de işini tamamladığı bilgisini ilgili TaskTracker'dan aldıktan sonra çalıştırdığı görevin durumunu başarılı olarak belirler. Bu durumda JobClient bu durumu kontrol ettiğinde başarılı olma durumunu ve diğer

istatistiksel bilgileri kullanıcıya sunar ve runJob() fonksiyonu sonlanmış olur. JobTracker tamamlanmış olan göreve ait yarattığı ortanca çıktı (intermediate output) gibi ara verileri temizler ve aynısını yapmaları için TaskTracker'ları da bilgilendirir. Hadoop ile MapReduce görevi başarıyla sonlandırılmış olur.



Şekil 3.8. HDFS'de işleyicilerin durum bilgilerinin aktarılma şeması [4]

4. GÖRÜNTÜ İŞLEME İÇİN HADOOP EKLENTİ MİMARİSİ

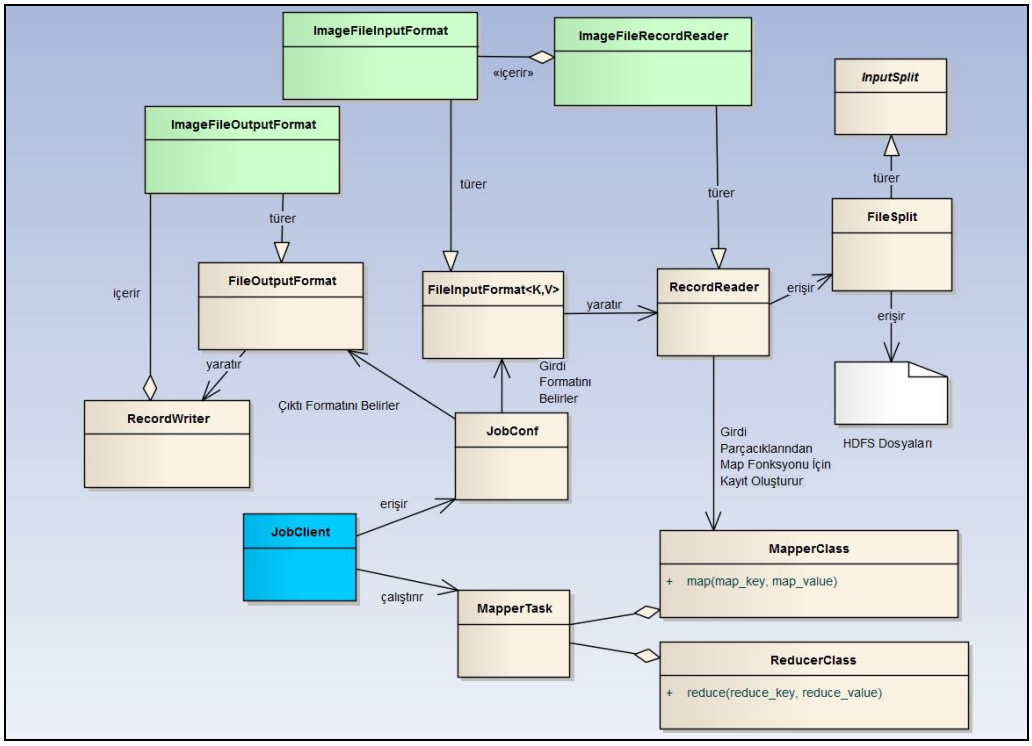
Hadoop'un yaygın kullanımına baktığımızda genelde web tabanlı verilerin dağıtık olarak kaydedilmesi ve kaydedilen veriler üzerinde aramalar yapılabilmesini sağlamak amaçlı olarak kullanıldığını görürüz. Bu şekilde elde edilen veriler HDFS'ye kaydedildiğinde indekslenerek ve sıkıştırılarak çok büyük boyutlu dosyalar halinde kaydedilmektedir. Hadoop çok büyük boyutlu dosyalar ile çalışmak için optimize edilmiştir [8]. Bunun nedeni FileInputFormat tarafından oluşturulan dosya parçacıklarının dosyaların paralel işlenmesi için farklı düğümlerdeki farklı işleyiciler tarafından kullanılmasıdır. Eğer dosya boyutu çok küçük olursa parçalanamaz (split) dolayısıyla dosyanın paralel işlenmesi mümkün olmamaktadır ve o dosya için sadece bir tane Map işleyicisi yeterli olmaktadır [5]. İmge dosyaları hem ikili (binary) formatta veriler olmaları, hem de boyut olarak küçük olmaları nedeniyle parçalanıp paralel işlenmemelidir. İmge dosyası parçalandığında bütünlüğü bozulur ve bilgisini kaybeder.

Dolayısıyla imgelerden bazıları, belirlenmiş Hadoop dosya parçacık boyutundan (split size) büyük boyutlu olsa bile yine de parçalanmaması gerekmektedir. Görüntü işleme algoritmasının tüm imge verisi üzerinde çalışması gerekmektedir. Bu şekilde tüm görüntü dosyasını parçalamadan Map İşleyiciye girdi yapabilmek için Hadoop'un içinde tanımlanan dosya girdi formatları yeterli olmamaktadır. Bu tez kapsamında gerçekleştirilen ImageFileInputFormat dosya girdi formatı bunu sağlamaya yönelik olarak Şekil 4.1.'de sergilendiği gibi tasarlanmış ve geliştirilmiştir.

Ayrıca map fonksiyonuna kayıt olarak verilmesi gereken yine tüm imge verisi olmalıdır ki map fonksiyonu imge verisi üzerinde görüntü işleme algoritmasını çalıştırabilsin. Bunu gerçekleştirmek için tez kapsamında ImageFileRecordReader kayıt oluşturucu sınıfı Şekil 4.1.'de sergilendiği gibi Hadoop sınıf yapısı içerisinde tasarlanmış ve geliştirilmiştir.

Görüntü işleme algoritması çalıştıktan sonra elde edilen çıktı eğer bir imge ise bunun tekrardan HDFS'ye bir imge olarak kaydedilmesi Hadoop'un kendi çıktı dosya

formatlarıyla mümkün değildir. Hadoop çalıştırdığı görevlere verdiği etiket numarasıyla tek bir çıktı dosyası oluşturur (örn: part-0001 gibi). Bu nedenle tez kapsamında denenen yüz saptama algoritmasında bulunan yüz imgelerinin HDFS'ye kendi isimleri ve koordinatlarını da içerecek şekilde isim ve formatını belirlenip kaydedilebilmesi için FileOutputFormat'tan türeyen ImageFileOutputFormat sınıfı Şekil 4.1.'de sergilenen Unified Modelling Language (UML) dili ile tasarlanmış ve geliştirilmiştir. Burada yeşil olan format sınıfları bu eklenti kapsamında geliştirilmiş olup sergilenen Hadoop'a ait diğer sınıfları da kendi içinde kullanarak HDFS'de imge okuma yazma ve işleme işlerini yönetebilmektedir.



Şekil 4.1. Geliştirilen Hadoop eklentisi UML tasarımı

Geliştirilen bu formatlar ve kayıt okuyucu sınıfı kullanılarak görüntü işleme amaçlı bir Hadoop java eklentisi geliştirilmiştir. Bu eklenti ile sadece imge verileri değil Hadoop ile diğer ikili (binary) formattaki dosyaların da işlenmesi mümkün olmaktadır.

4.1. Geliştirilen Hadoop Dosya Formatları ve Kayıt Okuyucu Sınıfı

4.1.1. ImageFileInputFormat

İşlenecek dosyanın parçalanmadan girdi olarak bir bütün olarak alınmasını sağlamanın birkaç yolu vardır. Bunlardan en kolayı; en küçük parçacık boyutunu en büyük parçacık boyut değeri (mapred.max.split.size) olarak atanmasıdır. Fakat bu yöntem ile parçalanması gerekebilecek dosyaların da parçalanması engellenmiş olur. Bunun yerine sadece imge işleme görevinde geçerli olacak bir yöntem denenmiştir. FileInputFormat sınıfından türemiş olan ImageFileInputFormat sınıfında dosyanın parçalanabilirlik özelliği kaldırılmıştır. Bu isSplittable fonksiyonu yeniden yazılarak gerçeklenmiştir.

Bu fonksiyon her dosya HDFS'den okunduğunda çağrılmakta ve geri dönüş değerine göre dosya parçalanarak InputSplit oluşturulmaktadır. Bu çalışma kapsamında HDFS'de bulunan imge dosyalarının parçalanmadan işlenmesi bu yöntemle sağlanmıştır.

4.1.2. ImageFileOutputFormat

Hadoop en yaygın olarak kullanılan ikili (binary) çıktı formatı SequenceFileOutputFormat sınıfıdır. Bu sınıf ile görev çıktısı anahtar değer ikililerinden oluşan kayıtlar halinde birleştirilerek tek bir SequenceFile ikili (binary) dosyası şeklinde HDFS'ye kaydedilir. SequenceFile formatında kaydedilen çıktı dosyasının ismi ve formatı Hadoop tarafından otomatik olarak belirlenir. Hadoop çalıştırdığı görevlere verdiği kimlik numaralarına göre çıktı dosyalarını isimlendirir. Bu tez kapsamında gerçeklenen imge işleme görevi HDFS'de çalıştırıldığında elde edilen çıktı dosyalarının imge dosyası olması beklenmektedir. Ayrıca elde edilen çıktı dosyalarının isimlendirilmesinde esnek bir yapı oluşturmak amaçlanmıştır.

Bu çalışma kapsamında gerçeklenecek yüz saptama algoritmasının çıktıları olan imgelere direkt erişimin olması için çıktıların imge formatında kaydı gerekmektedir. Saptanan yüzlere ait imgelerin isimlendirilmesinde de normal Hadoop çıktı dosyası isimlendirme yönteminin dışına çıkmak gerekmektedir. Hadoop içinde tanımlı olan

çıktı formatları bunu sağlamaya elverişli olmadığından FileOutputFormat sınıfından türeyen ImageFileOutputFormat sınıfı geliştirilmiştir.

Bu sınıf kapsamında ata sınıftan türetme sonucunda yeniden yazılması gereken en önemli fonksiyon getRecordWriter fonksiyonudur. Bu fonksiyon map fonksiyonu içerisindeki OutputCollector üzerinden çıktılar her yazılmak istendiğinde, kaydın çıktı dosyasına ne şekilde yazılacağının belirlendiği fonksiyondur. Bu çalışma kapsamında gerçeklenen map fonksiyonu içerisinde saptanan yüzlere ait veriler getRecordWriter ile dosyalara Şekil 4.2.'de sergilenen program kodundaki gibi yazılmaktadır. Burada FileSystem nesnesinden elde edilen girdi dosyası bilgileri de kullanılarak dosya formatı elde edilir ve istenen isim ve uzantıda dosya ayrı bir kanalla HDFS'ye ayrı bir dosya olarak kaydedilmiş olur. ImageFileOutputFormat sınıfı kullanılarak hazırlanan yüz saptama görevi sonucunda saptanan yüzlere direkt erişim mümkün olmaktadır. Bu yüz imgelerine erişmek için tekrardan çıktı dosyasını işleme tabi tutmak gerekmemektedir.

```
public RecordWriter<Text, BytesWritable> getRecordWriter( FileSystem ignored,
final JobConf job, final String name, Progressable progress) throws IOException{
        return new RecordWriter<Text, BytesWritable>() {
        public void write(Text key, BytesWritable value) throws IOException {
        Path file = FileOutputFormat.getTaskOutputPath(job,name);
        Path outputFolderPath = FileOutputFormat.getWorkOutputPath(job)
        FileSystem fs = file.getFileSystem(job);
        String filename = key.toString().substring(key.toString().lastIndexOf("/"));
        BinaryFileFormat.Writer out = BinaryFileFormat.createWriter (fs, job,
        filePathUnique, job.getOutputKeyClass(), job.getOutputValueClass());
        out.append(key,value);
        out.close();
        }
        public void close(Reporter reporter) throws IOException {/*out.close();*/}
    }
}
```

Şekil 4.2. Kayıt yazma fonksiyonunun program kodu

4.1.3. ImageFileRecordReader

İmge işleme uygulamasında map fonksiyonunun imge verisinin tüm içeriğini tek seferde işlemesi gerekmektedir. Özellikle yüz saptama algoritması resimdeki tüm yüzleri saptama üzerine kurulu bir algoritma olduğu için resmin tamamını tek seferde işler, bunu ancak tüm veriyi içeren tek bir kayıtla sağlayabilir.

Normalde ImageFileInputFormat imge verisinin tamamını işleyici için tek bir girdi parçacığı (input split) haline getirse de bu map fonksiyonunda imgenin bütün olarak işlenmesi için yeterli değildir. Map fonksiyonu girdi parçacığını direkt olarak almaz. Map fonksiyonu kayıt okuyucu (RecordReader) sınıfından gelen kayıtları girdi olarak alır. Normalde bir metin dosyasında her satır bir kayıt olabilirken, ikili (binary) dosyalar için kaydın sınırlarının belirlenmesi gerekmektedir. Bu nedenle imge içerisinde yüz saptama algoritmasını çalıştırabilmek için gerekli olan bütün imge verisini tek bir kayıt olarak sağlamaya yönelik ImageFileRecordReader sınıfı geliştirilmiştir.

```
public boolean next (NullWritable key, BytesWritable value) throws IOException {
    if(!processed) {
        byte[] contents = new byte[(int) fileSplit.getLength()];
        Path file = fileSplit.getPath();
        FileSystem fs = file.getFileSystem(conf);
        FSDataInputStream in = null;
        try{
            in = fs.open(file);
            IOUtils.readFully(in,contents,0,contents.length);
            value.set(contents,0,contents.length);
        }finally{
            IOUtils.closeStream (in);
        }
        processed = true;
        return true;
    } return false; }
```

Şekil 4.3. Kayıt oluşturan fonksiyonunun program kodu

RecordReader sınıfında tanımlı next() fonksiyonu kaydın oluşturulmasını ve map fonksiyonuna iletilmesini sağlayan fonksiyondur. Bu fonksiyon içerisinde bütün

imge verisini içeren FileSplit verisinin içeriği tek bir kayıt olacak şekilde bir seferde dosya sisteminden okunmuştur. Daha sonra kayıt oluşturma işleminin tamamlandığı bilgisi Şekil 4.3.'teki program kodunda sergilendiği gibi "processed" değişkeniyle tanımlanarak dosya okuma işlemi sonlandırılmıştır. Artık bu noktadan sonra map fonksiyonuna gelen kayıt bilgisi bir imgeye ait tüm veriyi içerecek şekilde oluşmuştur. Böylelikle map fonksiyonu kendi içinde yüz saptama algoritmasını bütün imge verisi üzerinde çalıştırabilir duruma gelmiştir.

4.2. Hadoop SequenceFile Dosya Formatı ve İkili Dosya İşleme

HDFS'de tutulan büyük boyutlu ikili (binary) formattaki dosyaları işlemek için en yaygın kullanılan format SequenceFile dosya formatıdır. Bir ikili dosya, işleyicilerde işlenmek için parçalanarak işleyicilere gönderilirse içeriğindeki ikili verinin işleyiciye giden kısmı anlamlı olmayabilir. Buna örnek olarak çok büyük boyutlu bir imge dosyası verisini verebiliriz. Bu imge dosyası HDFS'de kaydedilirken farklı bloklar halinde farklı düğümlere kaydedilir. Bu düğümlerde çalışacak işleyiciye bu parça girdi olarak verilirse işleyicinin bunu işlemesi mümkün olmamaktadır. Çünkü imgenin verisi belli bir formatta sıkıştırılmış bir veri olabilir. Dolayısıyla belli boyutta bir parçası işlenebilir olmamaktadır.

Bir başka örnek olarak da ağ trafiğinde akan verinin kaydedildiği büyük kütük dosyalarını verebiliriz. Bu dosyalar işlenmek için rastgele parçalandığında anlamsal olmayan parçacıklar oluşabilmektedir. Bu durumun önüne geçebilmek için büyük boyutlu ikili dosyaların, işlenmesinin anlamlı olacağı küçük parçalarının, ayrı kayıtlar halinde bir arşiv formatında tutulması gerekmektedir.

Hadoop içerisinde tanımlanan SequenceFile dosya formatı içerisinde <anahtar, değer> ikilileri şeklinde ikili formatta veri parçacıklarının işlenmeye hazır kayıtlar şeklinde tutulması mümkün olmaktadır. Hadoop SequenceFile formatındaki büyük boyutlu dosyaları kaydederken kayıt bütünlüğünü koruduğu için içerisindeki işlenebilen ikili veri parçacıkları bozulmadan işlenebilir hale gelmektedir.

SequenceFile dosya formatı ayrıca küçük boyutlu çok sayıda dosyanın HDFS'de kaydedilip işlenebilmesi için de bir arşiv dosya formatı olarak görev yapmaktadır. Bizim geliştirdiğimiz tekniklerden ikincisi olan imgeleri SequenceFile dosya

formatına dönüştürüp daha sonra işleme tekniğinde de bu özellik kullanılmıştır. Burada SequenceFileInputFormat dosya formatı ile okunan SequenceFile dosyası içerisindeki her bir kayıt direkt olarak map fonksiyonuna girdi olarak verilir.

Hadoop SequenceFile formatına dönüştürebilmek için genellikle önişleme yapılması gerekmektedir. Bu önişleme Hadoop ile bir SequenceFile'a dönüştürme görevi şeklinde olabileceği gibi. HDFS'ye dosyalar yazılmadan da bu işlem yapılabilmektedir. Genelde büyük boyutlu ikili formattaki dosyalar HDFS'ye yazılmadan önce işlenebilen alt parçacıkları birer kayıt olacak şekilde SequenceFile formatına dönüştürülür.

4.3. Hadoop Görevlerinin Mimarisi

Hadoop az sayıda çok büyük boyutlu dosya ile daha verimli ve hızlı bir şekilde çalışır bunun sebebi Hadoop'un çok büyük boyutlu dosyalarla çalışmak üzere tasarlanmış olmasıdır. Büyük boyutlu dosyaları HDFS'ye kaydederken belirlenmiş boyuttaki bloklar halinde farklı düğümlere kaydeder. Bu şekilde bu dosyalar üzerinde görevler çalıştırıldığında, her işleyici kendi düğümünde kayıtlı bloklardan girdi parçacıklarını alır ve işler. Bu durum da paralel işleme performansının çok iyi olmasının nedenidir. Görüntü dosyaları ise genellikle küçük boyutlu dosyalar olup çok fazla sayıda bulunabilirler. Bu durumda her bir görüntü için ayrı bir Map işleyici gerekeceğinden çok fazla sayıda Map işleyicisi oluşturmak ve bunları yönetmek gerekecektir. Bu nedenle, tez kapsamında yapılan çalışmada çok sayıda imge dosyasının HDFS'de paralel işlemek için iki farklı işleme tekniği geliştirilmiştir. Bu teknikler:

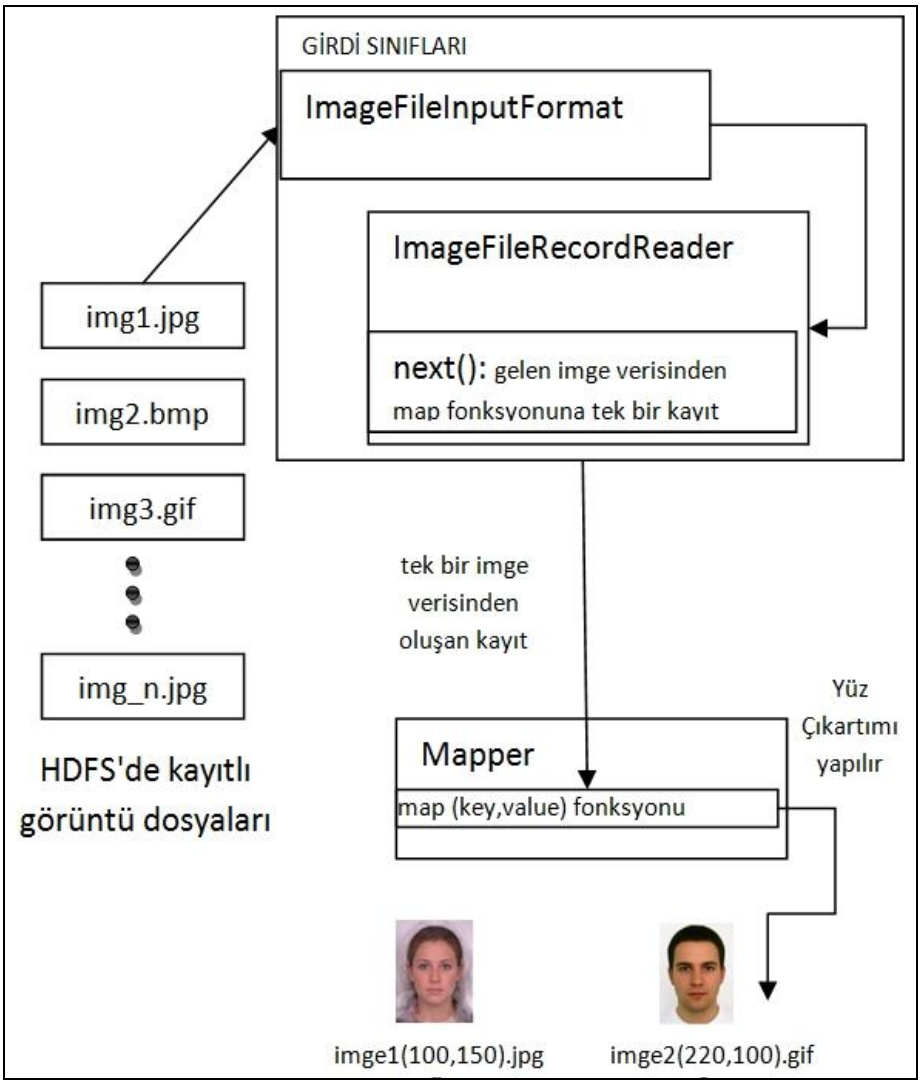
a.Her imge için bir işleyici kullanarak işleme tekniği.
b.İmgelerin birleştirilerek daha büyük dosyalar şeklinde HDFS'ye kaydedilmesi ve bu dosyaların işlenmesi tekniği.

4.3.1. Her imge için bir işleyici kullanma tekniği

Normalde imge boyutları küçük ve sayıları çok fazla olduğu için ve geliştirdiğimiz eklenti de imgelerin verisini tek bir kayıt olarak map fonksiyonuna verdiği için, her imge için bir Map işleyicisi şeklinde yüz saptama yapılmıştır. Bunun için

oluşturulan görevin girdi dosya formatı olarak eklenti kapsamında geliştirdiğimiz ImageFileInputFormat ve çıktı dosya formatı ImageFileOutputFormat sınıfları olmuştur.

Her imgeyi bir Map işleyicisi aldığından ve bu Map işleyicisi normalde ortanca çıktı (Intermediate Output) yerine direkt olarak HDFS'ye çıktı oluşturduğundan Reduce işleyicisine gerek duyulmamış ve görevin reduce sayısı sıfır olarak atanmıştır. Çıktı dosyaları HDFS'ye formatının korunması için sıkıştırılmadan kaydedilsin diye setCompressMapOutput(false) şeklinde sıkıştırma özelliği kapatılmıştır. İlgili görev oluşturulmuş ve farklı sayıda imge içeren girdi klasörlerinin içeriği paralel olarak işlenmiş ve Şekil 4.4.'te sergilenen sıraya uygun olarak saptanan yüz imgeleri belirtilen çıktı klasörüne kaydedilmiştir.



Şekil 4.4. Her imge için bir map işleyici tekniği çalışma şekli

4.3.2. İmgeleri birleştirme ve sonra işleme tekniği

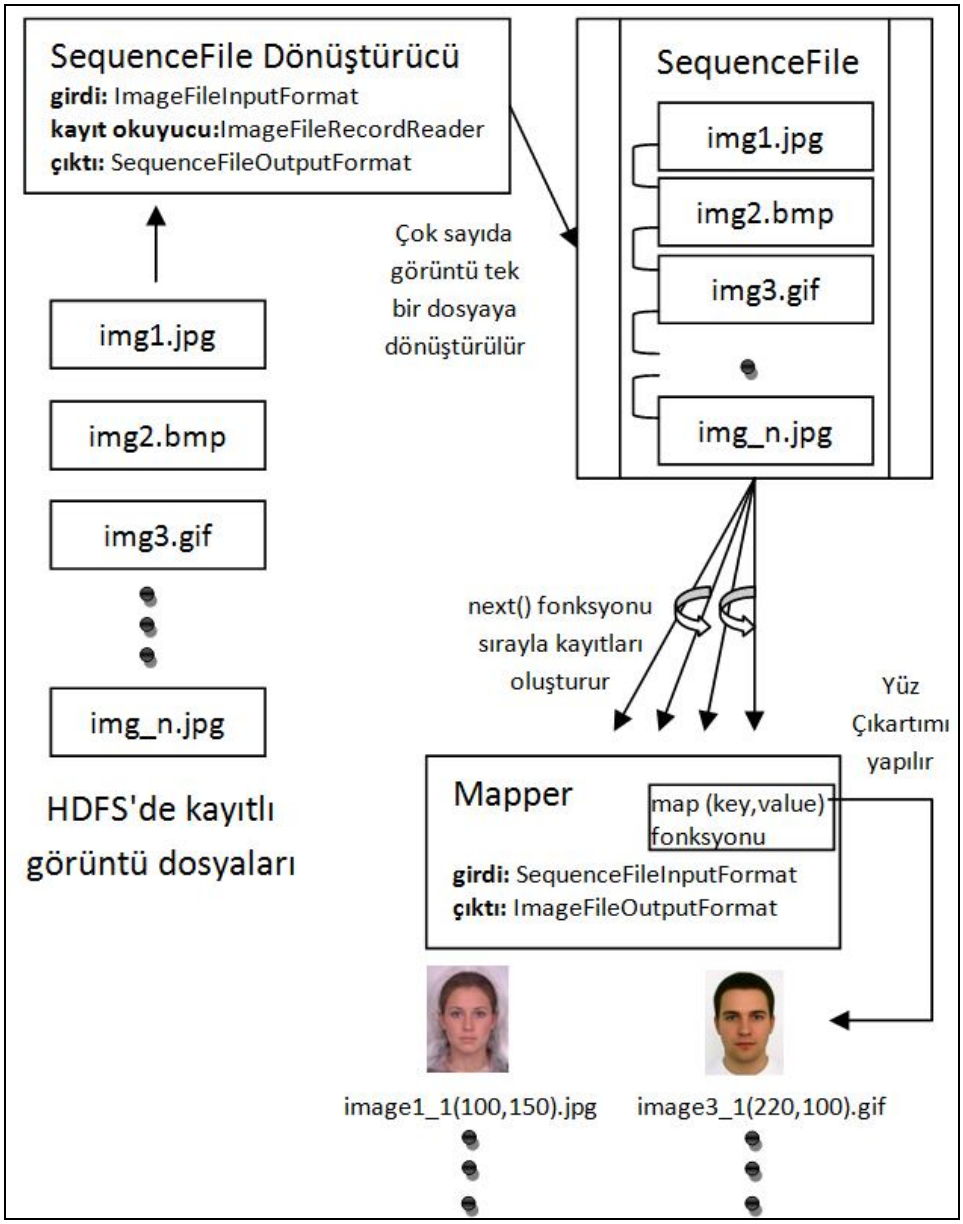
Her imge için bir Map işleyicisi tekniği her ne kadar gerek girdi gerek çıktı safhasında ön işleme veya sonradan işlemeye ihtiyaç duymayan ve direkt olarak imgeleri HDFS'den alarak işleyen bir teknik olsa da HDFS'nin çalışma tasarımına bu yapı uygun değildir. Bir başka ifadeyle Hadoop'taki NameNode yazılımı her bir kayıtlı dosya için bir metadata verisi tutmakta olduğundan çok büyük sayıda dosya NameNode'un yavaşlamasına ve gecikmelere neden olur [9].

Ayrıca her bir imge için bir işleyici oluşturulması aynı anda başlatılması ve yönetilmesi gereken çok sayıda işleyiciye neden olacağından sistemde JobTracker'ın bu kadar işleyiciyi sıralaması ve yönetmesi ciddi anlamda yavaşlar [5]. Bu probleme çözüm olarak HDFS'deki imgeleri işlemeye başlamadan önce birleştirerek çok büyük boyutlu bir arşiv dosyasına dönüştürmek ve daha sonra bu dosyayı işlemek tekniği kullanılabilir [8].

SequenceFile [7] Hadoop içerisinde tanımlı çeşitli dosyaları veya verileri belli kayıtlar şeklinde içinde depolaması için tasarlanmış büyük arşiv dosyalarıdır. İmgeleri birleştirerek bu dosyalara dönüştürmek ve daha sonra işlemek Hadoop uyumluluğu açısından en uygun yöntem olarak görülmüş ve bu çalışma kapsamında uygulanmıştır.

SequenceFile formatı kayıtları ikili (binary) <anahtar, değer> değerler şeklinde tutar. Bu MapReduce ile kullanıma çok uygun bir formattır çünkü parçalanabilir ve farklı düğümlere bu şekilde paylaştırılabilir. Ayrıca kendi içinde kayıt sırasında sıkıştırma özelliğine göre kaydedilecek verinin boyutunda azaltma da yapabilmektedir.

Hadoop ile paralel imge işleme çalışmamız kapsamında HDFS'de kayıtlı olan imgelerin birleştirilerek SequenceFile tipine dönüştürülmesi gerekmektedir. Bu işlemin yapılabilmesi içinde ayrı bir Hadoop görevi geliştirmek gerekmiştir. İkinci aşama olarak da SequenceFile tipine dönüştürülen imgelerin işlenebilmesi için SequenceFile işleyerek yüz imgelerini saptayan bir görev yazılması gerekmektedir. Şekil 4.5.'te bu tekniklerin çalışma şekli genel olarak sergilenmiştir.



Şekil 4.5. İmgeleri birleştirme ve sonra işleme tekniği çalışma şekli

4.3.2.1. İmgeleri birleştirerek SequenceFile formatına dönüştürme

İmgeleri birleştirerek SequenceFile'a dönüştürmek için yazılan görevde eklenti kapsamında geliştirmiş olduğumuz ImageFileInputFormat sınıfı ile girdi klasörlerinden girdi parçacıkları (InputSplit) oluşturulmuştur. Yine eklenti kapsamında geliştirilen ImageFileRecordReader kullanılarak map fonksiyonuna her bir imge verisi bir kayıt olarak verilmiştir.

Bu görevde map fonksiyonundan beklenen, kendisine gelen imge verisini SequenceFile'a bir kayıt olarak geçmektir. Bu nedenle Şekil 4.6.'daki program

kodunda sergilendiği gibi gelen veri üzerinde herhangi bir işlem yapmadan direkt olarak OutputCollector'a kayıt olarak iletmiştir. Burada collect() fonksiyonuna, imgenin adını anahtar değer olarak, imgenin içerik verisini de değer olarak aktarmıştır. Bu görevi oluştururken JobConf sınıfına Şekil4.7.'de sergilendiği gibi görev girdi çıktı formatları ve veri tipleri atanmıştır.

```
public void map(NullWritable key, BytesWritable value, OutputCollector
<Text, BytesWritable> output, Reporter reporter) throws IOException {
    String filename = conf.get("map.input.file");
    output.collect(new Text(filename), value); }
```

Şekil 4.6. Sequence dosya dönüştürme Map fonksiyonu program kodu

```
JobConf conf = new JobConf(seqconverter.class);
conf.setInputFormat(ImageFileInputFormat.class);
conf.setOutputFormat(SequenceFileOutputFormat.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(BytesWritable.class);
conf.setMapperClass(SequenceFileMapper.class);
```

Şekil 4.7. Sequence dosya dönüştürme JobConf ayarları program kodu

Bu görev, 2000 imge içeren bir klasör girdi olarak verilip HDFS üzerinde çalıştırıldığında Şekil 4.8.'de sergilenen SequenceFile tipinde dosya oluşmuştur:

Contents of directory /user/hadoop01/SEQ_OUT2000

Goto : /user/hadoop01/SEQ_OUT2(go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
part-00000	file	276.15 MB	3	64 MB	2012-01-19 20:43	rw-r--r--	hadoop01	supergroup

Şekil 4.8. Birleştirme sonucu oluşan SequenceFile'ın HDFS görünümü

4.3.2.2. SequenceFile formatındaki imgeleri işleme

Oluşturulan SequenceFile dosyası anahtar olarak imgenin adı ve değer olarak da imge verisini içeren kayıtlardan oluşan arşiv dosyaları şeklindedir. Bu dosyanın işlenmesi için girdi dosya formatı olarak SequenceFileInputFormat olarak atanmıştır. Çıktı formatı daha önceki teknikte olduğu gibi geliştirdiğimiz

ImageFileOutputFormat olarak Şekil4.9.'daki program kodunda sergilendiği gibi gerçeklenmiştir.

```
JobConf conf = new JobConf(imageprocessor.class);
conf.setInputFormat(SequenceFileInputFormat.class);
conf.setOutputFormat(ImageFileOutputFormat.class);
conf.set("mapred.output.compress", "false");
conf.set("mapred.compress.map.output","false");
conf.set("mapred.max.split.size","16777216");
conf.setCompressMapOutput(false);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(BytesWritable.class);
conf.setMapperClass(SequenceFileMapper.class);
conf.setNumReduceTasks(0);
SequenceFileInputFormat.setInputPaths(conf, new Path(inputFolderName));
ImageFileOutputFormat.setOutputPath(conf, new Path(outputFolderName));
```

Şekil 4.9. SequenceFile işleme JobConf ayarları program kodu

Önceki teknikten farklı olarak bu teknikte "mapred.max.split.size" değeri 16 megabyte olacak şekilde atanmıştır. Bunun sebebi varsayılan değer olan 64 megabyte ile imge arşivi işlenirken hem çok fazla hafıza gereksinimi doğmakta hem de eğer işleyici hata verirse tüm girdi parçacığının baştan işlenmesi gerekmektedir. TaskTracker her bir Map İşleyiciyi ayrı bir Java Sanal Makine (JVM) ile çalıştırır. Bu işleyicilerin çalışması sırasında JVM için belirlenen hafıza boyutunun üzerinde bir hafıza kullanımı olursa işleyici hata vermektedir. Bu durumda girdi parçacığının baştan işlenmesi gerekmektedir. Bu da görevin toplam çalışma süresini uzatacağından 16 megabyte gibi daha küçük bir parçacık boyutuna göre SequenceFile'ın parçalanıp işlenmesi uygulanmıştır.

SequenceFileInputFormat kendi içerisinde kayıt okuyucu sınıfı içermekte ve bu kayıt okuyucu sınıfı dosya içerisinde <anahtar, değer> ikilileri şeklinde bulunan imge verilerini, map fonksiyonuna iletmektedir. Bu nedenle bu kısım için fazladan bir kayıt okuyucu sınıfı geliştirilmesine gerek görülmemiştir.

Map fonksiyonu bir önceki teknikte map(NullWritable key, BytesWritable value,..) şeklinde anahtar parametresini dikkate almazken, SequenceFile içerisinde her bir

kayıda ait anahtar değeri dosya ismini içeren metin olduğundan fonksiyonun tanımı map (Text key, BytesWritable value,..) şeklinde kodlanmıştır.

Bölüm 5'te açıklandığı şekliyle OpenCV [6] kullanılarak, SequenceFile içerisindeki imgelerden saptanan yüzlere ait çıktılar bir önceki teknikte olduğu gibi geliştirilen ImageFileOutputFormat üzerinden HDFS'ye kaydedilmiştir.

5. ÖNERİLEN SİSTEM İLE YÜZ SAPTAMA DURUM ÇALIŞMASI

Geliştirilen imge işleme amaçlı Hadoop eklentisi, HDFS'de kayıtlı imge dosyalarını okuyarak Map işleyicileri içerisindeki map fonksiyonlarına girdi olarak verebilmektedir. Ayrıca elde edilen çıktı dosyalarını istenen format ve isimde HDFS'ye kaydını yapabilmektedir. Geliştirilen teknikle de imgeler işleme öncesi birleştirilerek SequenceFile dosya tipine dönüştürülerek Hadoop'un küçük boyutlu dosyalar ile çalışmasından kaynaklanan yavaşlıkların önüne geçilebilmektedir. Tüm bu geliştirmeler bu bölümde açıklanmıştır.

Map fonksiyonları kendilerine kayıt verisi olarak iletilen imge dosyalarında görüntü işleme algoritmasını çalıştırarak sonuçlarını elde etmektedirler. Bu tez kapsamında aşağıdaki bölümlerde açıklandığı şekilde imgeler üzerinde yüz saptama algoritması kullanılmıştır. Yüz saptama amaçlı ve geliştirilen teknikler kullanılarak Hadoop görevleri yazılmış, HDFS üzerinde bu görevler çalıştırılmış ve geliştirilen eklentinin ve tekniklerin etkinliği elde edilen sonuçlara göre değerlendirilmiştir.

5.1. İmge İşlemeye Yönelik Hadoop OpenCV Kütüphanesi Entegrasyonu

Günümüzde görüntü işleme amaçlı kullanılan pek çok yazılım geliştirme araçları ve kütüphaneleri kullanılmaktadır, bunlardan en yaygın kullanımı olan OpenCV [6] kütüphanesidir. C++ dilinde açık kaynak kodlu ve platform bağımsız olması ayrıca Intel Performance Primitives (IPP) [25] gibi hızlandırıcı algoritma kütüphaneleriyle de uyumlu çalışabilmesi açısından tercih edilmektedir.

Hadoop ise Java dili kullanılarak yazılmıştır. Geliştirici veya kullanıcı kendi görevini yazmak istediğinde Java dili ile Hadoop sınıflarını kullanması gerekmektedir. Bu tez kapsamında ilerde başka görüntü işleme algoritmalarını da Hadoop ile denemek açısından en kapsamlı görüntü işleme kütüphanesi olan OpenCV ile beraber kullanılması uygun görülmüştür. OpenCV C++ dilinde yazıldığı için Hadoop map fonksiyonları içerisinde ilgili algoritmalarına erişmek için Java Native Interface (JNI) [26] kullanılmasının gerekli olduğu görülmüştür.

Bu tez kapsamında asıl hedef görüntü işleme algoritmalarının Hadoop üzerinde paralelleştirilmesi ve bunun için Hadoop'un optimize edilmesidir. Bu nedenle OpenCV kütüphanesinin JNI ile kullanılabilir olmasını kendimiz kodlamak yerine bu konuda hazırlanan OpenCV İşleme ve Java kütüphanesini [27] Hadoop ile çalışacak şekilde güncelleme yoluna gidilmiştir.

Tüm map() fonksiyonu içeriği ve OpenCV ile JNI üzerinden nasıl entegre edildiği EK-A'da sunulan imageprocessor.java sınıfında incelenebilir. Şekil 5.1.'de JNI arayüzünü map fonksiyonu içerisinde nasıl kullandığımız sergilenmiştir.

```
PImage img = new PImage(imgWidth, imgHeight);
image_buffered.getRGB(0,0,imgWidth,imgHeight,img.pixels,0,imgWidth);
opencv.allocate(img.width,img.height);
opencv.copy(img);
opencv.cascade("FRONTALFACE_ALT",true);
faceRect = opencv.detect(true);
```

Şekil 5.1. JNI ile OpenCV kullanımı program kodu

Cascade() fonksiyonunun kullanımından anlaşılacağı gibi uygulamanın durum çalışması olarak yüz saptama yaptığımız için OpenCV içerisindeki Haar peşpeşe sınıflayıcı yüz saptama algoritması (Haar Cascade Face Detector) [6] kullanılmıştır. Haar özniteliklerine göre eğitilen bir sınıflandırıcıdır. Bu yüz saptama algoritması ile değişik boyutta ve açıda yüzleri imgelerin içinden saptamak mümkün olabilmektedir. Bu çalışma kapsamında ön yüz görünümü olan yüzler saptanmıştır. "FRONTALFACE_ALT" parametresi ile ön yüze göre eğitilmiş sınıflandırıcı kullanılır hale getirilmiştir.

Hadoop'ta java ile yazılan yeni sınıflar ve eklentiler, düğümler arasında paylaşılan JAR dosyasında yer alabilirken, dışarıdan kullanılan kütüphane ve diğer dosyaların da tüm düğümler tarafından çalışma zamanında erişilebilir olması gerekmektedir. Bunu sağlamak için her bir düğüme ilgili kütüphanenin kurulumunu yapmak hem kontrolü hem de yönetilmesi zor bir iştir. Hatalı kurulumlar yapılabilir veya kurulum yaparken sistemin başka bileşenlerine zarar verilebilir. Bunu önlemek için Hadoop dağıtık tampon hafıza alanı (Distributed Cache) özelliği kullanılmaktadır. Bir görev oluşturulurken Distributed Cache'de yer alacak dosyalar tampona eklenir. Bu

tampondaki dosyaları tüm düğümler çalışma sırasında kendi yerel hafıza alanlarına alarak kullanırlar.

Map fonksiyonu içerisinde JNI arayüzü üzerinden OpenCV kullanarak elde ettiğimiz yüzler, geliştirdiğimiz ImageFileOutputFormat sınıfı ile ayrı imgeler olarak HDFS'ye kaydedilmektedir. Bu kayıt sırasında normalde yüzün saptandığı imgenin adı "imge_aile.jpg" olsun ve bu imgede 4 tane yüz saptanmış olsun bunları HDFS'ye KaynakImge+SaptananYüzünNumarası+(BulunmaKoordinatı(x,y))+imgeFormatı şeklinde ve aşağıdaki gibi kaydedilmektedir (Örn: imge_aile1(x1,y1).jpg).

Bunu gerçeklemek ve bu isimdeki imgelerin verisini OutputCollector'a vererek HDFS'ye kaydetmek için Şekil 5.2.'deki program kodu gerçeklenmiştir:

```
for(int index = 0; index<faceRect.length;index++){
    BufferedImage faceImage = new BufferedImage (faceRect[index].width,
    faceRect[index].height, image_buffered.getType() );
    faceImage.setRGB(0, 0, faceRect[index].width, faceRect[index].height,
    opencv.getBuffer().pixels, imgWidth*(faceRect[index].y-1) +
    faceRect[index].x, imgWidth);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ImageIO.write(faceImage, "jpg", baos);
    BytesWritable imageDataOutput = new BytesWritable( baos.toByteArray());
    String imgName = filename.substring(0, filename.lastIndexOf("."));
    String imgFormatName =
    filename.substring(filename.lastIndexOf(".")+1,filename.length());
    String outputImgName =
    imgName + index+ "("+ faceRect[index].x +","+ faceRect[index].y+ ")"+ "." +
    imgFormatName;
    output.collect(new Text(outputImgName), imageDataOutput); }
```

Şekil 5.2. Saptanan yüzlerin HDFS'ye kaydedilmesi program kodu

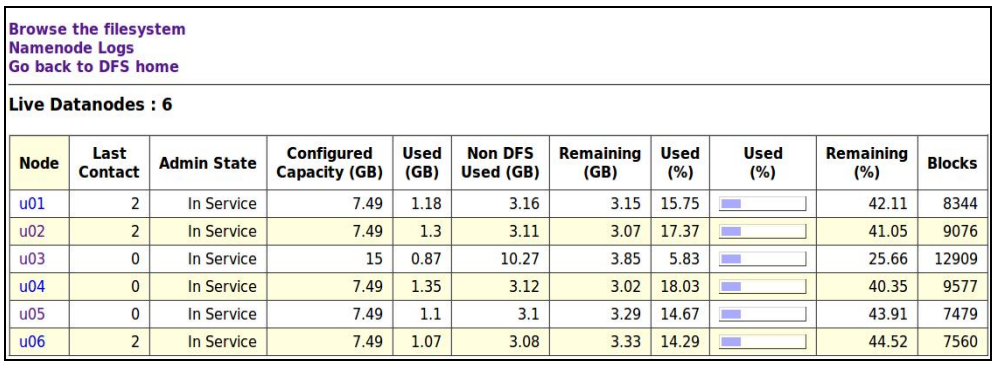
Burada OutputCollector'a ait collect() fonksiyonu içerisine HDFS'ye yazılacak dosya için kayıt bilgisi; anahtar olarak saptanan yüz dosyasının adı ve değer olarak saptanan yüzün verisi olacak şekilde çağrılmıştır.

5.2. Hadoop Görevlerinin Çalıştırılması ve Analizi

Hadoop ile görüntü işleme denemeleri Şekil 5.3.'te sergilenen altı düğüm'den oluşan bir dağıtık sistem ile yapıldı. Şekilde Hadoop yönetim aracının sunduğu web arayüzü görünmektedir. Bu yönetim aracı üzerinde HDFS'de bulunan görevlerin durumları, düğümlerin çalışma durumları ve kayıtlı dosya ve klasörlerin takibi yapılabilmektedir. Bu sayfada görülen altı düğümden "u03" etiketli düğüm yönetici düğüm olarak görev yapmaktadır. JobTracker ve NameNode yazılımları bu düğüm üzerinde çalışmaktadır. Bu nedenle işlemci sayısı, ayrılmış disk kapasitesi ve hafıza alanı miktarı yönetici düğüm için daha fazla tutulmuştur. Tablo 5.1.'de sistemin donanım bilgisi bulunmaktadır.

Tablo 5.1. Kullanılan donanım bilgisi

Donanım	Özellik
İşlemci	2.3Ghz Intel i7 2820QM 8MB Cache Yönetici düğüm için 2 çekirdek İşçi düğümler için 1 çekirdek ayrıldı
Hafıza	Yönetici Düğüm: 2GB İşçi Düğüm: 1GB
Sabit Disk Alanı	Yönetici Düğüm: 16GB İşçi Düğüm: 8GB
Ağ	VirtualBox Bridged JMicron Gigabit Ethernet Adaptor



Browse the filesystem
Namenode Logs
Go back to DFS home

Live Datanodes : 6

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
u01	2	In Service	7.49	1.18	3.16	3.15	15.75		42.11	8344
u02	2	In Service	7.49	1.3	3.11	3.07	17.37		41.05	9076
u03	0	In Service	15	0.87	10.27	3.85	5.83		25.66	12909
u04	0	In Service	7.49	1.35	3.12	3.02	18.03		40.35	9577
u05	0	In Service	7.49	1.1	3.1	3.29	14.67		43.91	7479
u06	2	In Service	7.49	1.07	3.08	3.33	14.29		44.52	7560

Şekil 5.3. Hadoop dağıtık sistemindeki makineler

Tablo 5.2.'de sergilendiği gibi düğümlerin her biri sanal makinelere (Virtual Machine) kuruldu. Sanal makine kullanımı genel performansta bir kayba neden olsa da kolay kurulumu ve yönetilmesi nedeniyle tercih edildi. Hadoop'ta işçi düğümlerin her biri aynı yazılımsal yapılandırmaya sahip olmalıdır. Bir başka ifadeyle, işçi

düğümlerde çalışan işleyiciler aynı yazılım birimlerini çalıştırırlar, bunlar map veya reduce fonksiyonlarını içeren JAR dosyalarıdır.

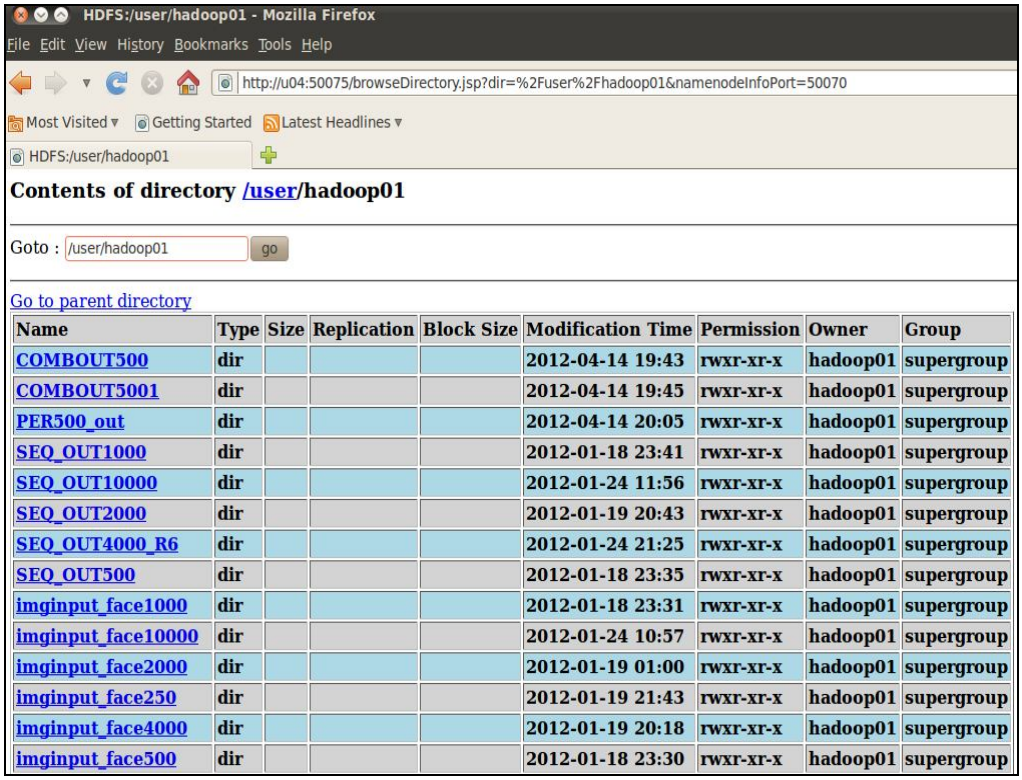
Tablo 5.2. Kullanılan yazılımlar

Yazılım	Versiyon
İşletim Sistemi	Ubuntu 10.04 LTS
HDFS	Hadoop 0.20.2
Java	JRE 1.6.0_26
OpenCV	OpenCV-2.3.0
Sanallaştırma Aracı	Oracle Virtual Box 4.1.8

Yüz saptama algoritması Map işleyicilerinde büyük miktarda görüntüyü işlerken bazen atanmış JVM dinamik hafıza alanını aşılabilmektedir. Bunu önlemek için Hadoop için JVM dinamik hafıza boyutu maksimum 600Mb olabilecek şekilde artırıldı. Oluşturulan çıktı imgeleri herhangi bir sıkıştırmaya tabi tutulmadan HDFS'ye kaydedildi.

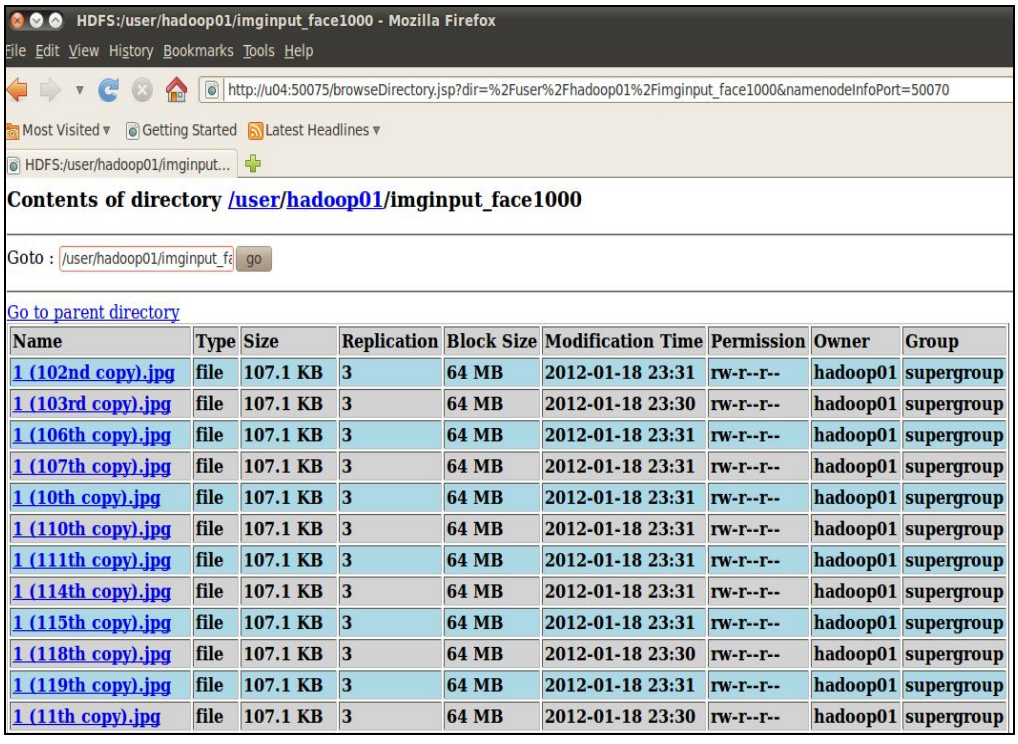
Görüntü işleme tekniklerinin başarımını görebilmek için farklı sayıda imge içeren klasörler girdi klasörleri olarak belirlendi. Kullanılan klasörlerin HDFS görünümleri Şekil 5.4.'te sergilenmektedir. Girdi verilerini içeren klasörlere beş farklı imge kopyalanarak klasörler arasında imge dosyalarının dağılım oranları korunmuştur. Klasörlerin içerdiği imgeler Şekil 5.5.'te görülebileceği gibi HDFS'de orijinal formatlarını korumaktadırlar.

Girdi klasörlerindeki imgeleri işlemek için yukarıda bahsedilen yöntemler kullanılarak iki ayrı HDFS görevi yazıldı. Her görüntü için bir işleyici yaklaşımı ve SequenceFile ile işleme yaklaşımı kullanılarak Hadoop görevleri JAR dosyası formatında hazırlandı ve Şekil 5.6.'da sergilenen biçimde çalıştırıldı. Komut satırının en üstünde çalıştırılacak JAR dosyasının adı ("faceextractor_combine10.jar") ve girdi olarak alacağı girdi klasörünün ismi ("imginput_face500") verilerek "bin/hadoop" komutuyla çalıştırılmaktadır.



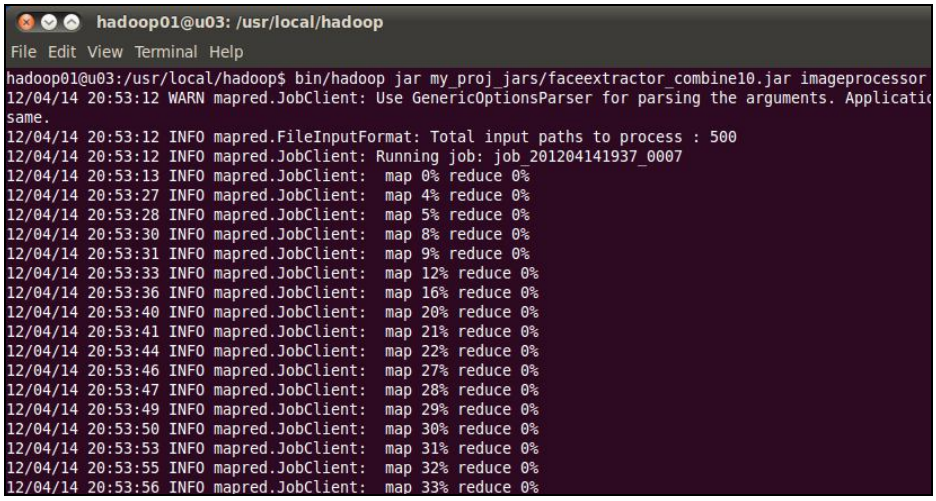
Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
COMBOUT500	dir				2012-04-14 19:43	rwxr-xr-x	hadoop01	supergroup
COMBOUT5001	dir				2012-04-14 19:45	rwxr-xr-x	hadoop01	supergroup
PER500_out	dir				2012-04-14 20:05	rwxr-xr-x	hadoop01	supergroup
SEQ_OUT1000	dir				2012-01-18 23:41	rwxr-xr-x	hadoop01	supergroup
SEQ_OUT10000	dir				2012-01-24 11:56	rwxr-xr-x	hadoop01	supergroup
SEQ_OUT2000	dir				2012-01-19 20:43	rwxr-xr-x	hadoop01	supergroup
SEQ_OUT4000_R6	dir				2012-01-24 21:25	rwxr-xr-x	hadoop01	supergroup
SEQ_OUT500	dir				2012-01-18 23:35	rwxr-xr-x	hadoop01	supergroup
imginput_face1000	dir				2012-01-18 23:31	rwxr-xr-x	hadoop01	supergroup
imginput_face10000	dir				2012-01-24 10:57	rwxr-xr-x	hadoop01	supergroup
imginput_face2000	dir				2012-01-19 01:00	rwxr-xr-x	hadoop01	supergroup
imginput_face250	dir				2012-01-19 21:43	rwxr-xr-x	hadoop01	supergroup
imginput_face4000	dir				2012-01-19 20:18	rwxr-xr-x	hadoop01	supergroup
imginput_face500	dir				2012-01-18 23:30	rwxr-xr-x	hadoop01	supergroup

Şekil 5.4. Girdi klasörlerinin HDFS görünümleri



Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
1 (102nd copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (103rd copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:30	rw-r--r--	hadoop01	supergroup
1 (106th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (107th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (10th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (110th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (111th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (114th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (115th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (118th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:30	rw-r--r--	hadoop01	supergroup
1 (119th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:31	rw-r--r--	hadoop01	supergroup
1 (11th copy).jpg	file	107.1 KB	3	64 MB	2012-01-18 23:30	rw-r--r--	hadoop01	supergroup

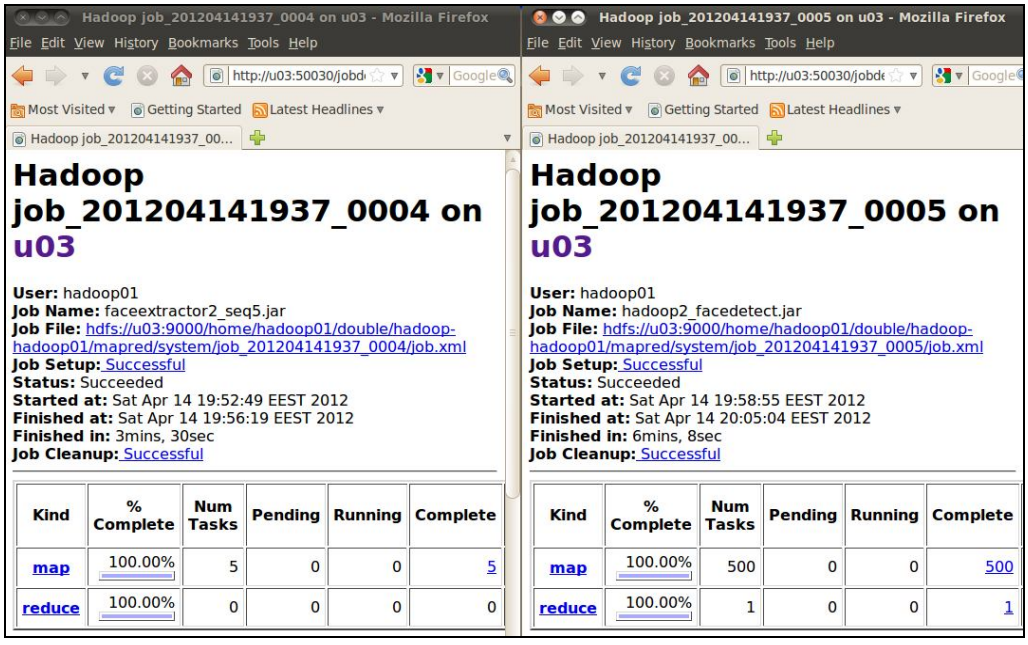
Şekil 5.5. Girdi klasörlerinde bulunan imgelerin HDFS görünümü



Şekil 5.6. Göreve ait JAR dosyasının Hadoop'ta çalıştırılması

SequenceFile işleme yaklaşımı için görüntü klasörlerinin içeriği önişleme yapıldı ve SequenceFile'a dönüştürüldü daha sonra Hadoop görevi çalıştırıldı. Çalışma süreleri web tabanlı HDFS yönetim aracından Şekil 5.7.'deki gibi her görevin sonunda okunarak kaydedildi. Sol tarafta SequenceFile işleme tekniği ile geliştirilen görevin 500 imge girdisi ile tamamlanma bilgileri sergilenmiştir. Sağ tarafta da her imgeye bir işleyici tekniğine ait görevin tamamlanma bilgileri sergilenmiştir.

Hadoop görevi ile ilgili web arayüzünde sergilenen parametrelerin anlamları Tablo 5.3'te ifade edilmiştir.

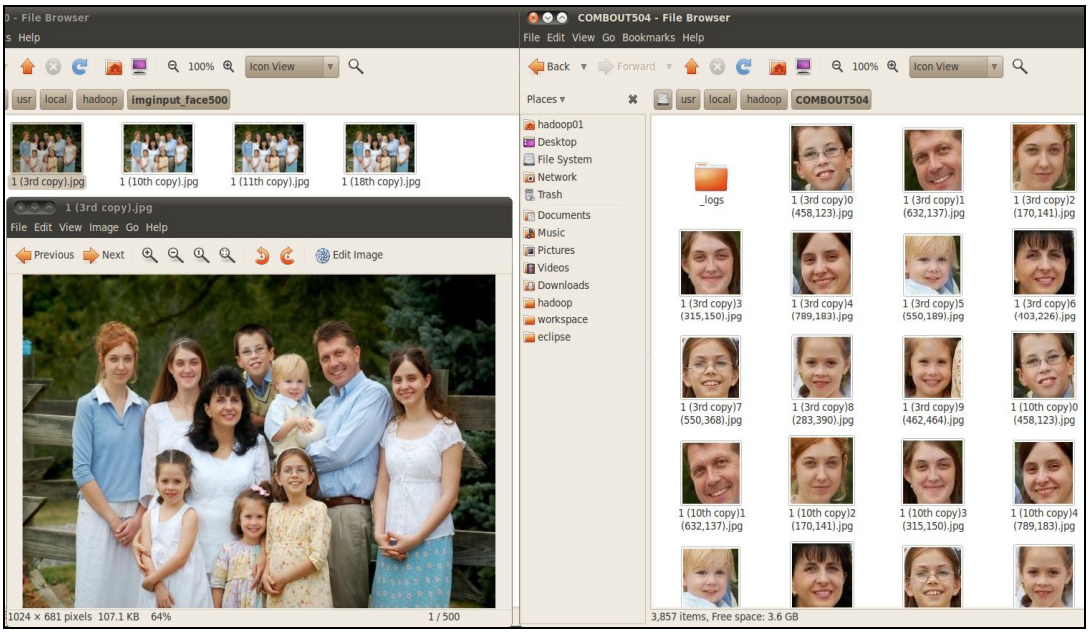


Şekil 5.7. Görevlerin durum bilgilerinin sergilenmesi

Tablo 5.3. Hadoop yöneticisi Web arayüzündeki parametreler

Parametre	Açıklaması
User	Görevi çalıştıran kullanıcı adı
Job Name:	Görevi içeren JAR dosyası adı
Job File:	Göreve ait görev çalışma bilgilerini içeren Extensible Markup Language (XML) formatındaki dosya
Job Setup:	Görevin başarılı bir şekilde sistemde başlatıldığını gösteren durum bilgisi
Status:	Görevin sonlanma durum bilgisi
Started/Finished At:	Görevin başlama ve tamamlanma zamanları
Finished In:	Görevin toplam tamamlanma süresi
Job Cleanup:	Görevin tamamlandıktan sonra oluşan yardımcı yerel dosyaların (ortanca çıktı gibi) temizlenme durum bilgisi
MapReduce Tablosu:	Oluşturulan MapReduce işleyicilerin toplam sayısını ve tamamlanma yüzdelerini sergiler

Görevler çalıştırıldıktan sonra her iki görevde de ortak ImageFileOutputFormat kullanıldığı için saptanan yüzler HDFS'ye aynı format ve isimlendirmeyle kaydedilmiştir. Şekil 5.8.'de girdi resmi ve görevin sonlanması sonucu oluşan çıktı klasörü içeriği görülebilir.



Şekil 5.8. Görevlerde kullanılan girdi imgesi ve saptanan yüzler

Şekil 5.3.'te sergilenmiş olan, altı adet düğümden oluşan HDFS bilgisayarları üzerinde her iki teknik ve geliştirilen eklenti kullanılarak hazırlanan Hadoop görevleri çalıştırılmıştır. Her görevin en doğru tamamlanma süresini elde edebilmek için aynı görev aynı girdi imgeleriyle birkaç kez çalıştırılarak ortalama tamamlanma süreleri ve standart sapma değerleri elde edilmiştir. Tablo 5.4'te her imge için bir işleyici görevinin farklı sayıda imge içeren girdi klasörlerindeki tamamlanma süreleri sergilenmiştir. Burada imge sayısı arttıkça oluşan işleyici sayısı da aynı oranda artmaktadır. Ayrıca 7000 ve üstü sayıda imge içeren girdi klasörlerinde çalışma süresi çok zaman aldığı için sadece 2 kez çalıştırılarak sonuçlar elde edilmiştir.

Tablo 5.4. Her imge için bir işleyici görevinin performans değerleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	1000	5	0.11	6.15
2000	2000	7	0.12	11.7
3000	3000	7	0.13	17.35
4000	4000	6	0.17	23.36
5000	5000	7	0.29	29.73
6000	6000	6	0.34	35.64
7000	7000	5	0.51	41.36
8000	8000	8	0.64	47.33

SequenceFile'a dönüştürme görevinde imgeler herhangi bir işleme tabi tutulmayıp sadece arşivlenme yapıldığı için bu önişleme sürecine ait tamamlanma süreleri Tablo 5.5'te sergilendiği gibi olmuştur. SequenceFile'a dönüştürme görevinde 6 Map işleyici ve 1 Reduce işleyici görev almış ve bu şekilde SequenceFile formatında çıktı dosyası oluşturulmuştur. Farklı çalıştırmalarda aynı girdi imge sayısı için çok yakın tamamlanma süreleri çıkmış bu nedenle standart sapma küçük değerler şeklinde oluşmuştur. SequenceFile'a dönüştürme, imgeleri HDFS'ye kaydederken yapılacak bir önişleme görevidir. İmge işleme gibi hesap yoğun bir görev değildir. Bu nedenle de tamamlanma süreleri kısa çıkmıştır.

Tablo 5.6.'da SequenceFile şekline dönüştürülmüş imgeleri işleme görevinin farklı sayıda imge içeren girdi klasörlerindeki tamamlanma süreleri sergilenmiştir. İmgeleri SequenceFile oluştuktan sonra SequenceFile işleme görevi çalıştırılmış ve aşağıdaki

sonuçlar elde edilmiştir. SequenceFile işleme esnasında oluşan Map işleyici sayısı belli dosya büyüklüklerinde Hadoop tarafından yük dengeleme (load balancing) yapmak için yeniden ayarlanmıştır. Bunda ayrıca 16 MB olarak atanmış olan "mapred.max.split.size " parametresinden kaynaklı SequenceFile boyutu büyüdükçe oluşan girdi parçacığı sayısının artması da etkili olmuştur. Genel olarak bu işleyici sayısı artışı SequenceFile işleme yönteminde gerçekleşen tamamlanma sürelerinin girdi imge adediyle doğrusal olarak artışının korunmasını sağlamıştır.

Tablo 5.5. SequenceFile'a dönüştürme görevinin performans değerleri

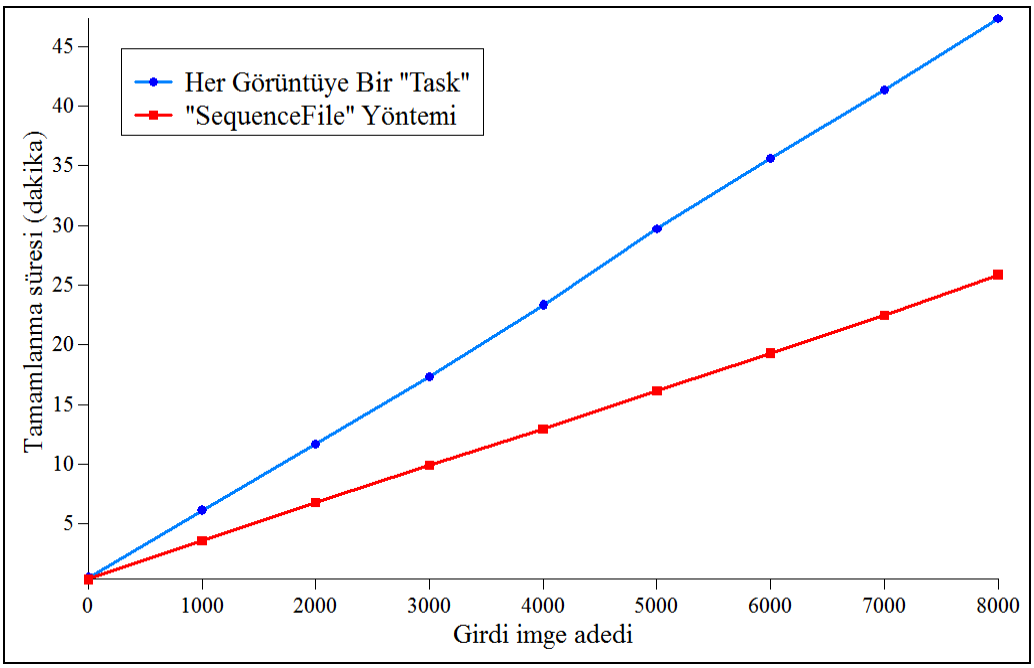
İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	7	0	0.01	0.31
2000	7	0	0.02	0.63
3000	7	0	0.01	0.94
4000	7	0	0.02	1.24
5000	7	0	0.03	1.55
6000	7	0	0.02	1.85
7000	7	0	0.03	2.16
8000	7	0	0.02	2.45

Tablo 5.6. SequenceFile işleme görevinin performans değerleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	36	5	0.07	3.6
2000	36	5	0.10	6.81
3000	36	5	0.12	9.91
4000	36	5	0.13	12.96
5000	36	6	0.15	16.15
6000	36	6	0.16	19.3
7000	36	8	0.22	22.47
8000	36	5	0.30	25.88

Şekil 5.9.'da elde edilen tüm bitirme süreleri, girdi imge adetlerine göre ölçülerek grafiksel olarak sergilenmiştir. Bu grafiğe göre imgeleri SequenceFile formatına

dönüştürdükten sonra işleme tekniğinin, Hadoop'ta imge işlemede ciddi performans artışı sağladığı görülebilmektedir.



Şekil 5.9.Tekniklerin yüz saptama görevlerini tamamlama performans grafiği

Yukarıda çalışma sonuçları sergilenen her iki görevde de girdi imgeleri üzerinde yüz saptama algoritması çalıştırılarak bulunan yüz bilgileri HDFS'ye sonuç olarak yazılmıştır. Elde edilen çalışma zamanlarının büyük bir kısmını imgeler üzerinde çalışan görüntü işleme algoritmasının kullandığı zaman oluşturmaktadır. Burada tekniklerin sadece dosyaların HDFS'den okumak ve map işleyicilerine ileterek girdi oluşturmaya yönelik performanslarını karşılaştırmak için görüntü işleme kodu çıkartılarak görevler yeniden oluşturulup Şekil 5.3.'te sergilenen bilgisayarlar üzerinde çalıştırılmıştır.

Görüntü işleme kısmını iptal etmek için map fonksiyonu içerisinde OpenCv işlemlerinin yapıldığı program kod satırları iptal edilmiştir. Bu durumda her imge için bir işleyici yönteminde HDFS'den okunan imge yine ImageFileInputFormat ve ImageFileRecordReader sınıflarından geçerek map fonksiyonuna girdi olarak iletilmiş fakat burada imge işleme yapılmadan direkt olarak çıktı oluşturulmuştur. SequenceFile işleme yönteminde de imgeler SequenceFile içerisinden okunarak map fonksiyonuna iletilmiş fakat burada yüz saptama işlemleri gerçeklenmeden çıktı oluşturulmuştur. Tablo 5.7.'de her imge için bir işleyici tekniği kullanılarak yüz

saptama yapılmadan imgelerin HDFS'de map fonksiyonlarına iletilerek isim bilgilerinin çıktı olarak kaydedilmesi görevinin çalışma süreleri verilmiştir. Tablo 5.8.'de ise SequenceFile formatına dönüştürülmüş imgelerin map fonksiyonuna girdi olarak verilmesi ve yüz saptama işlemi yapılmadan çıktının oluşturulmasına ait performans zaman bilgileri sergilenmiştir.

Tablo 5.7. Yüz saptama yapmadan her imge için bir işleyici görevlerinin performans değerleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	1000	5	0.11	4.56
2000	2000	7	0.13	8.55
3000	3000	7	0.17	12.9
4000	4000	6	0.27	17.41
5000	5000	7	0.39	21.82
6000	6000	6	0.41	26.33
7000	7000	5	0.40	30.92
8000	8000	8	0.45	35.72

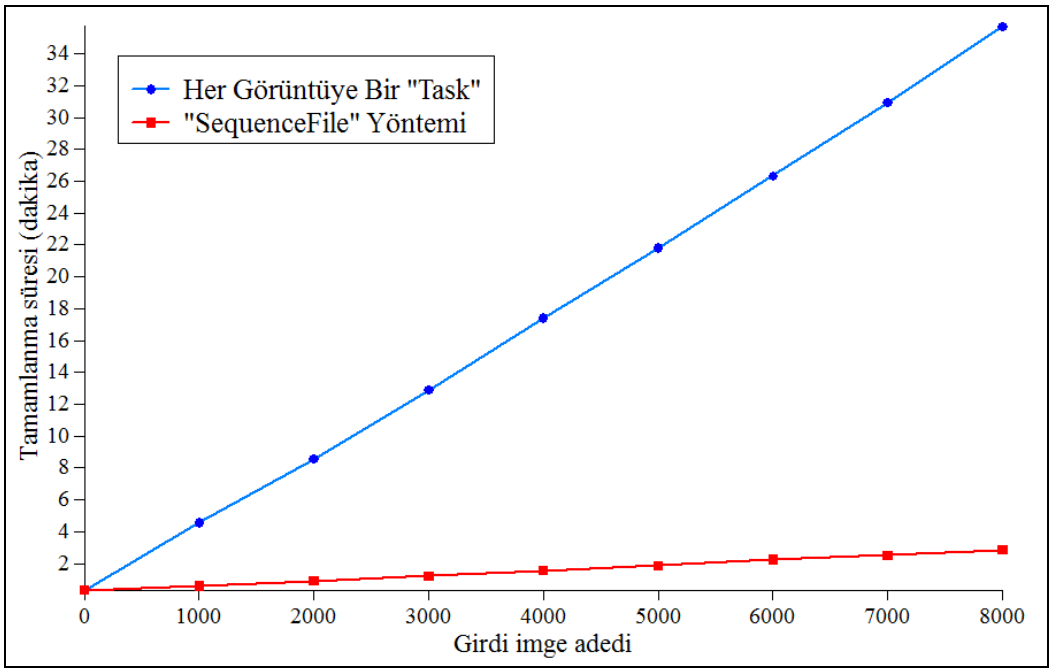
Tablo 5.8. Yüz saptama yapmadan SequenceFile işleme görevlerinin performans değerleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	36	0	0.13	0.61
2000	36	0	0.15	0.91
3000	36	2	0.20	1.23
4000	36	5	0.21	1.56
5000	36	5	0.22	1.88
6000	36	5	0.28	2.25
7000	36	7	0.31	2.53
8000	36	6	0.42	2.85

Yukarıdaki tablolarda her iki teknik kullanılarak elde edilen görevlerin tamamlanma sürelerini tek bir grafik üzerinde görüntülemek istersek Şekil 5.10.'u elde ederiz. Burada görevlerin tamamlanma zamanlarını Şekil 5.9.'daki verilerle karşılaştırdığımızda, SequenceFile işleme tekniğinin %90 daha az zaman harcayarak

görevi tamamladığı görülmektedir. Bunun sebebi SequenceFile tekniğinde işleyicileri başlatma ve yönetme yükünün çok az olması, asıl iş yükünü imge işlemenin oluşturmuş olmasıdır. Map işleyiciler içerisinden de imge işleme kısmı iptal edildiğinden toplam çalışma süresi %90 oranında kısalmıştır.

Her görüntü için bir işleyici kullanma tekniği çalışma zamanlarını karşılaştırdığımızda Şekil 5.9.'daki duruma göre yaklaşık %25 daha kısa sürede görevlerin tamamlandığı görülmektedir. Bunun nedeni: her imgenin Hadoop işleyicisi ile işlenmesi sırasında geçen Hadoop yapılandırma ve işleyicileri sıralama işleminin bu teknikte çok fazla zaman almasıdır. Her nekadar imge işleme kısmı map fonksiyonlarında iptal edilmiş olsa da bu toplam çalışma süresinde sadece %25'lik bir kısalmaya neden olmuştur.



Şekil 5.10. Yüz saptama yapmadan görevlerin tamamlanma performans grafiği

5.2.1. Görevlerin gerçek makinelerde çalıştırılması

Önceki bölümde sanal makinalar üzerinde kurulmuş olan 6 bilgisayardan oluşan Hadoop dağıtık dosya sisteminde çalıştırılan görevlere ait sonuçlar sergilenmiştir. Sanallaştırmanın kurulum ve yönetim kolaylıkları sağlaması yanında sistem kaynaklarını kullanmakta performans kaybı olabilmektedir. İşlemci kullanımı ve disk kullanımında meydana gelen bu performans kayıplarının Hadoop görevlerinin

çalışma sürelerini etkileyeceği de öngörülmektedir. Geliştirilen SequenceFile formatında birleştirilmiş imge arşivi işleme ve her imge için bir işleyici kullanma tekniklerinin performansının sanallaştırma yapılmadan ortaya konması için 2 makineye Tablo5.9.'da özellikleri belirtilen sistem kurulmuş ve görevler bu makineler üzerinde çalıştırılmıştır.

Tablo 5.9. Gerçek makinelerin donanım bilgisi

1.Bilgisayar	2.Bilgisayar
2.3-3.2Ghz Intel i7 2820QM İşlemci	2.53Ghz Intel P8700 İşlemci
8 GB Bellek	4 GB Bellek
500GB Disk Alanı	160GB Disk Alanı
100 MBit Ethernet Kartı	100 MBit Ethernet Kartı

Kullanılan yazılımların versiyonları bir önceki sanal makinalarda kullanılan ve Tablo 5.2.'de sergilenen şekilde ayarlanmıştır. Aynı görevler aynı girdi klasörleri üzerinde çalıştırılmıştır. Her imge için bir işleyici tekniğiyle geliştirilen görev çalıştırılmış ve Tablo 5.10.'daki tamamlanma süreleri elde edilmiştir. SequenceFile şeklinde arşivlenmiş imgeler üzerinde yüz saptama görevi çalıştırılmış ve Tablo 5.11.'deki sonuçlar elde edilmiştir.

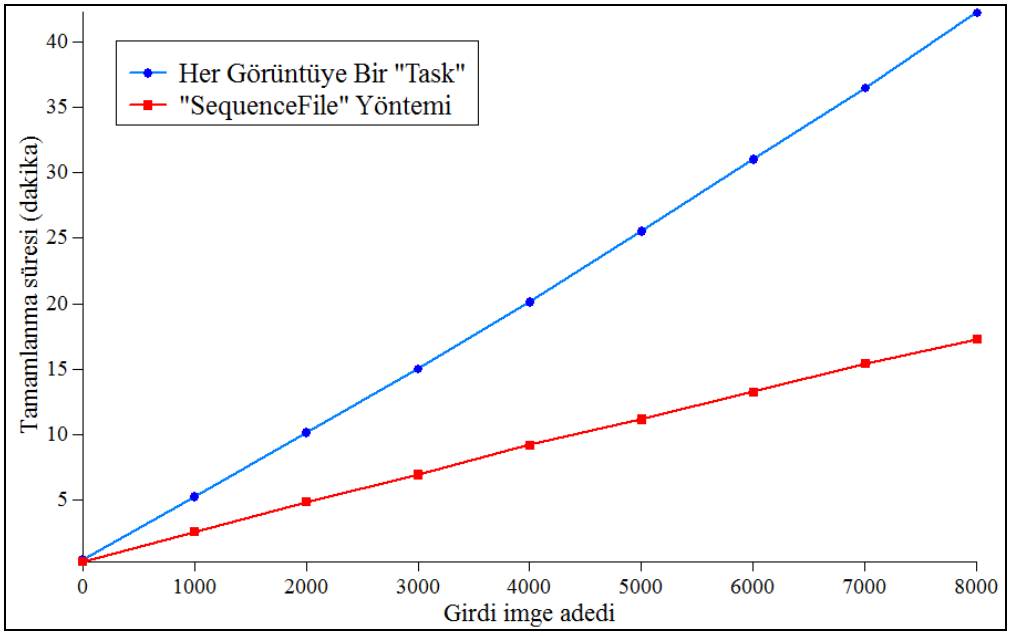
Tablo 5.10. Gerçek makinelerde her imge için bir işleyici görevinin tamamlanma süreleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	1000	5	0.13	5.27
2000	2000	7	0.20	10.18
3000	3000	7	0.21	15.04
4000	4000	6	0.28	20.13
5000	5000	7	0.30	25.54
6000	6000	6	0.34	31.02
7000	7000	5	0.41	36.45
8000	8000	8	0.64	42.22

Tablo 5.11. Gerçek makinelerden elde edilen SequenceFile işleme görevinin tamamlanma süreleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	36	5	0.1	2.61
2000	36	5	0.18	4.88
3000	36	5	0.22	6.98
4000	36	5	0.40	9.21
5000	36	6	0.6	11.2
6000	36	6	1.36	13.27
7000	36	8	1.43	15.43
8000	36	5	1.84	17.28

Bu performans sonuçlarının grafiksel gösterimi Şekil 5.11.'de sergilenmiştir.



Şekil 5.11. Gerçek makinelerde tekniklerin yüz saptama görevlerini tamamlama performans grafiği

Şekil 5.9.'da aynı görevler sanal makina üzerinde çalıştırılmakta ve toplamda aynı sayıda task ile işlemler tamamlanmaktaydı. Şekil 5.11.'de sergilenen sonuçlar ile karşılaştırdığımızda: Sanal makine kullanımına göre SequenceFile işleme tekniğinde %30 civarında bir hızlanma olurken. Her imge için bir işleyici tekniğinde %10 civarı bir hızlanma olmuştur. Bunun sebebi her imge için bir işleyici tekniğinde dosya sayısı kadar çok sayıda işleyicinin yaratılması ve

yönetilmesi için artan bilgisayar ağ haberleşmesi yükünü gösterebiliriz. SequenceFile işleme tekniğinde yaratılan işleyici sayısı az olduğundan bu yükün yansıması daha az olmuş, performans artışı daha fazla olarak görülebilmiştir.

Sanal makinalarda yaptığımız gibi gerçek makinalarda da yüz saptama algoritmasını iptal ederek görevleri yeniden çalıştırdığımızda Tablo 5.12 ve Tablo 5.13.'deki görevlerin tamamlanma performans sonuçlarını elde ederiz.

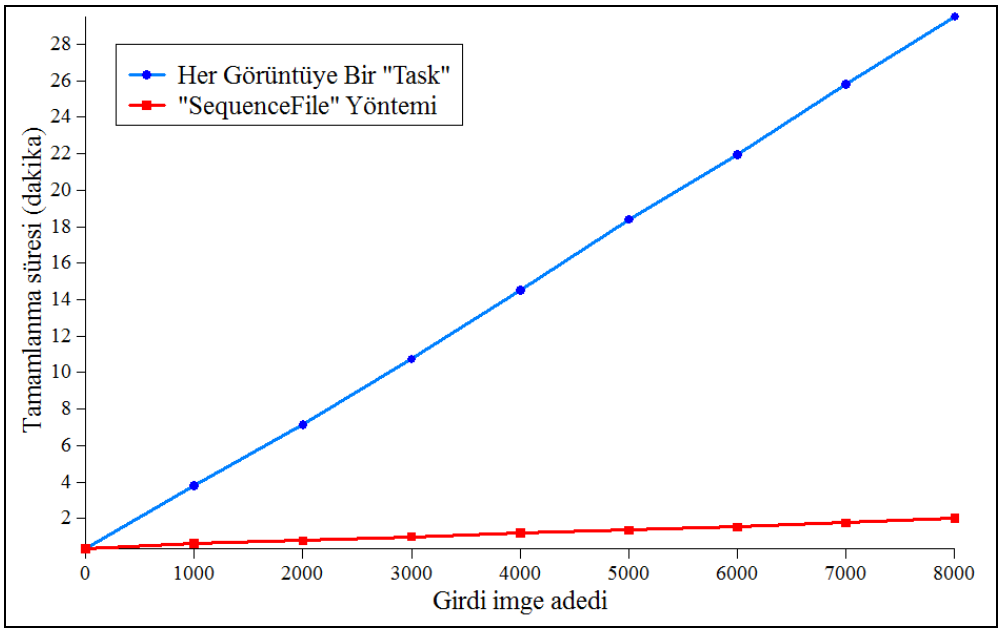
Tablo 5.12. Gerçek makinelerde yüz saptama yapmadan her imge için bir işleyici tekniği performans değerleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	1000	3	0.14	3.81
2000	2000	4	0.22	7.13
3000	3000	4	0.30	10.75
4000	4000	3	0.40	14.51
5000	5000	5	0.42	18.4
6000	6000	4	0.56	21.96
7000	7000	4	0.62	25.81
8000	8000	6	0.73	29.52

Tablo 5.13. Gerçek makinelerde yüz saptama yapmadan SequenceFile işleme tekniğiyle performans değerleri

İmge Sayısı	Oluşan İşleyici Sayısı	Ortalama Başarısız İşleyici Sayısı	Standart Sapma (dk)	Tamamlanma Süresi (dk)
1000	36	0	0.03	0.65
2000	36	0	0.07	0.78
3000	36	0	0.11	1
4000	36	1	0.10	1.2
5000	36	2	0.13	1.36
6000	36	2	0.14	1.53
7000	36	2	0.20	1.76
8000	36	2	0.19	2

Bu tablolardaki sonuçlarını Şekil 5.12.'de grafiksel olarak görmekteyiz.



Şekil 5.12. Gerçek makinelerde tekniklerin yüz saptama yapmadan görevlerin tamamlanma performans grafiği

Şekil 5.10.'da sanal makinalar üzerinde çalıştırılan, yüz saptama yapılmadan çalışan görevlerin sonlanma zamanları ile gerçek makinalarda elde ettiğimiz sonuçları karşılaştırdığımızda, gerçek makina kullanımında hızlanmanın olduğunu görmekteyiz. Her imge için bir işleyici tekniği ile geliştirilen görevde %15 gibi bir hızlanma olurken, SequenceFile işleme görevinde %30 civarında bir hızlanma olmuştur. SequenceFile işleme görevinde daha fazla hızlanma olmasının sebebi bu teknikte az sayıda işleyici yaratılmakta ve az sayıda işleyici ile ağ üzerinden haberleşme yükü dolayısıyla daha az olmaktadır. Her imge için bir işleyici görevinde ise çok sayıda işleyici ile haberleşmek ve bu işleyicilerin her birini başlatıp sonlandırmak için sürekli bir ağ haberleşmesi gerekmekte bu da performans artışının istenen miktarın altına düşmesine sebep olmaktadır.

5.2.2. Tekniklerin performanslarının karşılaştırılması

İki ayrı teknik uygulanarak elde edilen sonuçlara göre bu çalışmada geliştirilen I/O dosya formatları ve kayıt okuyucu sınıflarıyla beraber SequenceFile kullanımı, küçük görüntü dosyalarını işlemede ciddi performans artışı sağlayan bir yöntem olarak ortaya çıkmıştır. Küçük boyutlu dosyaları SequenceFile'a dönüştürüp daha sonra

işleme tekniği hem Şekil 5.9.'daki hem de Şekil 5.10.'daki durumda her imge için bir işleyici tekniğine çok daha hızlı çalıştığı görülmüştür. Şekil 5.10.'da sergilenen grafik, imgelerin map işleyicilere iletilmesi fakat map fonksiyonu içerisinde yüz saptama yapılmadan sonlanması durumunun performansını sergilemektedir. İmge işleme, işlemci yoğun bir operasyon olduğundan Şekil 5.9.'daki SequenceFile işleme işleme görevlerinin tamamlanma süreleri Şekil 5.10.'da yaklaşık %90 kısaldığı görülmüştür. Her imge için bir işleyici tekniğiyle geliştirilen görevler de ise hızlanma aynı ölçüde olmayıp %25 civarında olmuştur. Bunun sebebe bu teknikte oluşturulan çok sayıda işleyicinin yönetimi ve yapılandırılması çok fazla işlem gerektirmektedir ve yavaşlamaya neden olmaktadır.

SequenceFile kullanımının en başarılı çözüm olmasının nedenleri şunlardır: (1) SequenceFile'da oluşturulan dosya sayısı daha azdır, bu da NameNode üzerindeki yükü azaltır [8]. (2) Belirlenen SplitSize'a göre hadoop az sayıda işleyici yaratmıştır. Az sayıda işleyici olduğundan JobTracker ve TaskTracker'lar üzerindeki görev yönetim yükü azalmıştır. (3) işleyici sayısı az, aldıkları girdi verisi büyük olduğundan, işleyici başına tek seferde daha fazla görüntü işlemesi yapılmıştır. "Her Görüntüye bir Task" görevinin tamamlanma süresi dosya sayısı arttıkça doğrusal olarak artmıştır. SequenceFile yönteminin süre artış eğrisinin eğimi de dosya sayısı arttıkça aynı kalmıştır. Belli noktalarda doğrusallıkta kırılmalar olmasının sebebi her görevi çalıştırmada farklı sayıda işleyicinin başarısız olabilmesidir. Bir göreve ait işleyicilerden biri başarısız olduğunda JobTracker bu işleyiciyi tekrardan çalıştırır, bu durum da toplam çalışma süresinde artmaya neden olabilmektedir. SequenceFile işlemede işlenen parçacık boyutu büyük olduğundan; işleyicinin başarısız olması görevin toplam tamamlanma süresinde daha fazla sapmaya neden olabilmektedir. Ayrıca bir işleyici bazen başlar başlamaz başarısız olurken bazen de tamamlanmasına yakın başarısız olabilir. Bu da direkt olarak başarısız işleyici sayısına göre doğrusallıktan sapma kestirimini güçleştirmektedir. SequenceFile işlemede işleyiciler için dosya başına gerekli olan başlatma, yönetme ve sonlandırma sürelerinde (bookkeeping overhead) azalma olduğundan görevlerin toplam tamamlanma süreleri ciddi anlamda kısalmıştır. SequenceFile kullanımı hem sanal makinalarda hem de gerçek makinalar üzerinde çalıştırılan yüz saptama görevlerinde diğer tekniğe göre 2 kata yakın hızlanma sağlamıştır.

6. SONUÇLAR VE ÖNERİLER

Hadoop'un günümüzde kullanılan en yaygın MapReduce ile paralel hesaplama sağlayan dağıtık dosya sistemi oluşu, Hadoop'a farklı teknolojilerin uyarlanmasını çok değerli kılmaktadır. Bu tez kapsamında Hadoop'u büyük boyutlu imge işleme noktasında kullanılabilir kılmak için gerekli eklenti tasarımı ve geliştirmesi yapılmıştır. Ayrıca Hadoop'un büyük boyutlu dosyalarla en hızlı şekilde çalışması için tasarlanmış olmasın fakat bizim çalışmamızda imge gibi küçük boyutlu dosyaları işlememizden kaynaklı yavaşlık problemini aşmak için imgeleri birleştirerek işleme tekniği de geliştirilmiştir.

Bu çalışma sırasında Hadoop'un yeni bir teknoloji olması nedeniyle özellikle yaygın kullanımının (metin tabanlı arama eşleştirme) dışındaki kullanımlar hakkında fazla detaylı bilgiye ulaşamamaktan kaynaklı zorluklar yaşanmıştır. Ayrıca kurulumu ve yönetilmesi diğer dağıtık sistemlere göre kolay olsa da özellikle OpenCV [6] gibi farklı kütüphanelerle çalışır hale getirme noktasında başlangıçta ciddi sıkıntılar yaşanmıştır. JNI [26] ile OpenCV tümleştirme yapılması da özellikle hafıza yönetimi ve imgelerin dönüştürülmesi açısından çok zor olmuştur.

Hadoop imge işleme eklentisi geliştirilmesinde, Hadoop açık kaynak kodu alınıp bu kod üzerinde uyumluluğun sağlanması için pek çok denemelerin yapılması gerekmiştir. Geliştirilen Hadoop girdi ve çıktı formatlarının tasarımı geliştirilmesi ve denenmesi noktasında kaynak sıkıntısı da olmuştur. Ayrıca Hadoop'ta çalışması için geliştirilen kodun denenmesi ve çıkan hataların bulunması dağıtık sistemde çok zor olmuştur. Bu problemin çözümü için kütük dosyalarına mesaj yazdırarak kodda hata bulma ve giderme yöntemi uygulanmıştır.

Özellikle imgeleri birleştirerek işleme tekniği ile ciddi performans kazanımı elde edilmiş olsa da bu teknikte imgelere direkt olarak ulaşılamaması ve ön işleme ihtiyacının olması nedeniyle bazı uygulamalar için bu yöntem uygun olmayabilir. Bu çalışmaya ek olarak ilerde imgeleri gruplayarak ama birleştirmeden işleyicilere

gönderme yöntemi geliştirilip imgeler üzerinde uygulanabilir. Bu durumda biraz daha performans artışı ve kullanım kolaylığı ortaya çıkabilir.

Çalışmanın ülkemizde bu alanda yapılmış az sayıda çalışmalardan biri olması nedeniyle başka birçok çalışma için bir çıkış noktası ve başvuru kaynağı olabileceği temennisini taşımamaktayız.

Tez çalışmasının ana katkıları aşağıdaki gibi maddeler halinde sıralanabilir;

a. Hadoop MapReduce programlama modelinin mimarisi açıklanmıştır
b. Hadoop dağıtık dosya sisteminin yapısı, bileşenler ve MapReduce işleyicileriyle olan ilişkileri açıklanmıştır.
c. Hadoop sisteminde küçük boyutlu dosya kullanımından kaynaklanan problemler açıklanmış ve bunlara çözümler sunulmuştur.
d. Hadoop ile imge veritabanları üzerinde paralel görüntü işleme için Hadoop için girdi ve çıktı dosya format sınıfları ve kayıt okuyucu sınıfları geliştirilmiş ve açıklanmıştır
e. Hadoop ile görüntü işleme için en yaygın kullanılan ve C++ dilinde kodlanmış olan OpenCV kütüphanesinin Hadoop'a tümleştirmesi JNI arayüzü ile yapılmış ve açıklanmıştır.
f. Hadoop ile her bir imge için bir Map işleyici yaratma tekniği geliştirilen Hadoop eklentisi ile gerçeklenmiş ve farklı girdi klasörleri üzerinde çalıştırılarak sonuçları kaydedilmiştir.
g. Hadoop ile imgeleri birleştirip SequenceFile formatına dönüştüren bir görev yazılmış ve girdi klasörleri bu dönüşüme tabi tutulmuştur.
h. İmge dosyalarından oluşan SequenceFile dosyalarını işlemek için Hadoop görevi yazılmış ve çalıştırılarak sonuçları kaydedilmiştir.
i. Her iki geliştirilen tekniğe göre görevlerden elde edilen sonuçlar karşılaştırılarak Hadoop eklentisi ile geliştirilen SequenceFile'a dönüştürme ve işleme yönteminin başarısı açıklanmış ve sergilenmiştir.
j. Ulusal ve uluslararası büyük görüntü işleme projelerinde kullanılabilecek bir altyapı oluşturulmuş, Hadoop sistemi üzerinde gerekli testleri yapılarak bu çalışma kapsamında tasarımı ve geliştirilme aşamaları açıklanmıştır.

İleride bu çalışmada elde edilen yüzler üzerinde yüz tanıma (face recognition) yapılabilmesi için gerekli Hadoop eklentisinin hazırlanması planlanmaktadır. Ayrıca video içerisinde karakter ve yüz tespiti gibi konularında kullanılabilmesi için video işlemeye yönelik eklenti yazılması planlanmaktadır. Bu şekilde özellikle güvenlik amaçlı kamera kayıtlarının toplandığı merkezlerde kişi araması gibi projelerde altyapı olarak kullanılabilir olacaktır. Sonuç olarak Hadoop ile görüntü işleme amaçlı geliştirilen bu eklenti kapsamında, büyük miktarda imge üzerindeki yüz saptama işlemlerini paralelleştirmek verimli ve hızlı bir şekilde gerçeklenmiştir.

KAYNAKLAR

[1] http://hadoop.apache.org/ (Ziyaret tarihi: 1 Ocak 2012).

[2] Ghemawat S., Gobioff H., Leung S., "The Google File System,"Proc. of the 19th ACM Symp. on Operating System Principles, 29–43, 2003.

[3] Dean J. and Ghemawat S., "MapReduce: Simplified data processing on large clusters," Communications of the ACM, 2008, **51**, 107–113.

[4] White T., Hadoop: The Definitive Guide. O'Reilly Media, Inc.June 2009.

[5] http://www.cloudera.com/blog/2009/02/02/ the-small-files-problem/ (Ziyaret tarihi: 1 Mayıs 2012).

[6] http://opencv.willowgarage.com/wiki (Ziyaret tarihi: 1 Şubat 2012).

[7] http://wiki.apache.org/hadoop/SequenceFile (Ziyaret tarihi: 1 Şubat 2012).

[8] Dong B., Qiu J., Zheng Q., Zhong X., Li J., Li Y., "A Novel Approach to Improving the Efficiency of Storing and Accessing Small Files on Hadoop: A Case Study by PowerPoint Files," Services Computing (SCC), 2010 IEEE International Conference, 65-72, 2010.

[9] Liu X., Han J., Zhong Y., Han C. and He X., "Implementing WebGIS on Hadoop: A Case Study of Improving Small File I/O Performance on HDFS," Proc. of the 2009 IEEE Conf.on Cluster Computing, DOI: 10.1109/CLUSTR.2009.5289196.

[10] Carns P. H., Ligon W. B., Ross R. B. and Thakur R., Pvfs: A parallel file system for linux clusters. In In Proceedings of the 4th Annual Linux Showcase and Conference, 317–327, 2000.

[11] Honeyman P., Hildebrand D. and Ward L., Large files, small writes and pnfs. In in Proceedings of the 20th ACM International Conference on Supercomputing, 116–124, 2006.

[12] http://www.gluster.org/ (Ziyaret tarihi: 1 Mart 2012).

[13] Vaidya M., Parallel Processing of Cluster by Map Reduce, International Journal of Distributed and Parallel systems (IJDPS), 2012, **3**, 167-179.

[14] Wiley K., Connolly A., Krugho S., Gardner J., Balazinska M., Howe B., Kwon Y. and Bu Y., Astronomical Image Processing with Hadoop, Astronomical Data Analysis Software and Systems XX. ASP Conference Proceedings, 2010.

[15] Qiu X., Ekanayake J., Beason S., Gunarathne T., Fox G., Barga R. and Gannon D., "Cloud technologies for bioinformatics applications," in Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers. ACM, 1-10, 2009.

[16] Isard M., Budiu M., Yu Y., Birrell A., Fetterly D., "Dryad: Distributed data-parallel programs from sequential building blocks," European Conference on Computer Systems, 59-72, 2007.

[17] Krishna M., Kannan B., Ramani A., Sathish S. J., Implementation and Performance Evaluation of a Hybrid Distributed System for Storing and Processing Images from the Web. Cloudcom, 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 762-767, 2010.

[18] Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra T., Fikes A. and Gruber R., Bigtable: a distributed storage system for structured data. In OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, 15–15, 2006.

[19] http://cassandra.apache.org/ (Ziyaret tarihi: 1 Şubat 2012).

[20] Chen S.. Cheetah: A high performance, custom data warehouse on top of mapreduce. In VLDB, 1459–1468, 2010.

[21] Stonebraker M., Abadi D., D. Dewitt J., Madden S., Paulson E., Pavlo A., Rasin A.. Mapreduce and parallel dbmss: friends or foes? Commun. ACM, 2010, **53**, 64–71.

[22] http://lucene.apache.org/ (Ziyaret tarihi: 1 Şubat 2012).

[23] http://nutch.apache.org/ (Ziyaret tarihi: 1 Mayıs 2012).

[24] Courier: the remote procedure call protocol. Xerox System Integration Standard XSIS-038112, Xerox Corporation, Stamford, Connecticut, 1981.

[25] http://software.intel.com/en-us/articles/intel-ipp/ (Ziyaret tarihi: 1 Şubat 2012).

[26] http://java.sun.com/docs/books/jni/ (Ziyaret tarihi: 1 Ocak 2012).

[27] http://ubaa.net/shared/processing/opencv/ (Ziyaret tarihi: 1 Şubat 2012).

EKLER

Ek-A

Hadoop Eklentisi Kapsamında Geliştirilen Sınıflara Ait Program Kodları

Bu ekte, Hadoop ile büyük ölçekli imge işlemeye yönelik geliştirilmiş eklentiye ait sınıfların Java dilinde geliştirilmiş program kodları sunulmuştur.

ImageFileInputFormat ve ImageFileRecordReader Sınıfları:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reporter;

class ImageFileRecordReader implements RecordReader<NullWritable,
BytesWritable> {
    private FileSplit fileSplit;
    private Configuration conf;
    private boolean processed = false;
    public ImageFileRecordReader(FileSplit fileSplit, Configuration conf)
    throws IOException {
    this.fileSplit = fileSplit;
    this.conf = conf;
}
@Override
public NullWritable createKey() {
    return NullWritable.get();
}
@Override
    public BytesWritable createValue() {
    return new BytesWritable();
}
@Override
public long getPos() throws IOException {
    return processed ? fileSplit.getLength() : 0;
}
@Override
public float getProgress() throws IOException {
    return processed ? 1.0f : 0.0f;
```

```
}
@Override
public boolean next(NullWritable key, BytesWritable value) throws IOException {
    if (!processed) {
        byte[] contents = new byte[(int) fileSplit.getLength()];
        Path file = fileSplit.getPath();
        FileSystem fs = file.getFileSystem(conf);
        FSDataInputStream in = null;
    try {
        in = fs.open(file);
        IOUtils.readFully(in, contents, 0, contents.length);
        value.set(contents, 0, contents.length);
    } finally {
        IOUtils.closeStream(in);
    }
        processed = true;
        return true;
    }
    return false;
}
@Override
public void close() throws IOException {
    // do nothing
    }
}

public class ImageFileInputFormat extends FileInputFormat<NullWritable,
BytesWritable> {

protected boolean isSplitable(FileSystem fs, Path filename) {
    return false;
}

@Override
public RecordReader<NullWritable, BytesWritable> getRecordReader(
InputSplit split, JobConf job, Reporter reporter) throws IOException {
// TODO Auto-generated method stub
    return new ImageFileRecordReader((FileSplit) split, job);
    }
}
```

ImageProcessor Hadoop Görevi Kodu:

```
import java.awt.Image;
import java.awt.Component;
import java.awt.Toolkit;
import java.awt.image.BufferedImage;
import java.awt.image.MemoryImageSource;
```

```
import java.awt.Rectangle;

import java.io.File;
import java.io.InputStream;
import java.net.URI;
import java.net.URISyntaxException;
import java.nio.ByteOrder;
import java.util.List;
import java.util.Scanner;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.imageio.stream.ImageInputStream;
import javax.imageio.stream.ImageOutputStream;
import javax.imageio.stream.MemoryCacheImageOutputStream;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.lib.IdentityReducer;
import org.apache.hadoop.mapred.lib.MultipleSequenceFileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class imageprocessor extends Configured implements Tool {

static class SequenceFileMapper extends MapReduceBase implements
Mapper<NullWritable, BytesWritable, Text, BytesWritable> {
private JobConf conf;
@Override
public void configure(JobConf conf) {
    this.conf = conf;
}
@Override
public void map(NullWritable key, BytesWritable value,OutputCollector<Text,
BytesWritable> output, Reporter reporter)throws IOException {
```

```
    String path = System.getProperty("java.library.path");
    System.setProperty("java.awt.headless","true");
    System.out.println(path);

    String filename = conf.get("map.input.file");
    byte[] bytes2 = value.getBytes();
    ImageInputStream in = ImageIO.createImageInputStream(new
    ByteArrayInputStream(bytes2));
    BufferedImage image_buffered = ImageIO.read(in);
    int imgWidth = image_buffered.getWidth();
    int imgHeight =image_buffered.getHeight();
    Rectangle[] faceRect;
    OpenCV opencv;
    Rectangle[] squares = new Rectangle[0];
    // OpenCV setup
    PImage img = new PImage(imgWidth, imgHeight);
    image_buffered.getRGB(0,0,imgWidth,imgHeight,img.pixels,0,imgWidth);
    opencv.allocate(img.width,img.height);
    opencv.copy(img);
    opencv.cascade("FRONTALFACE_ALT",true);
    faceRect = opencv.detect(true);

    for(int index = 0; index<faceRect.length;index++){
        BufferedImage faceImage = new BufferedImage( faceRect[index].width,
        faceRect[index].height,image_buffered.getType());
        faceImage.setRGB (0, 0, faceRect[index].width, faceRect[index].height,
        opencv.getBuffer().pixels,
         imgWidth*(faceRect[index].y-1)+faceRect[index].x, imgWidth);
         ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ImageIO.write(faceImage, "jpg", baos);
        BytesWritable imageDataOutput = new BytesWritable( baos.toByteArray());
        String imgName = filename.substring(0, filename.lastIndexOf("."));
        String imgFormatName = filename.substring (filename.lastIndexOf (".") + 1,
        filename.length());
        String outputImgName = imgName +index+ "("+ faceRect[index].x +","+
        faceRect[index].y+ ")" +"."+imgFormatName;
        System.out.println(outputImgName);
        output.collect(new Text(outputImgName), imageDataOutput);
    }


    }
}
@Override
public int run(String[] args) throws IOException {
    JobConf conf = new JobConf(imageprocessor.class);
    Scanner input = new Scanner(System.in);
//JobBuilder.parseInputAndOutput(this, getConf(), args);
if (conf == null) {
```

```
    return -1;
}
String inputFolderName;
String outputFolderName;
if(args.length<2){
    inputFolderName = "imginput_face";
    outputFolderName = "face_01";
}else{
    inputFolderName = args[0];
    outputFolderName = args[1];

}
conf.setInputFormat(ImageFileInputFormat.class);
conf.setOutputFormat(ImageFileOutputFormat.class);
conf.set("mapred.output.compress", "false");// compression yapmayacaï¿½ï¿½z.
conf.set("mapred.compress.map.output","false");
conf.setCompressMapOutput(false);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(BytesWritable.class);
conf.setMapperClass(SequenceFileMapper.class);
conf.setNumReduceTasks(0);
ImageFileInputFormat.setInputPaths(conf, new Path(inputFolderName));
ImageFileOutputFormat.setOutputPath(conf, new Path(outputFolderName));

try {
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-android-arm.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-linux-x86.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-linux-x86_64.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-windows-x86.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-windows-x86_64.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-macosx-x86_64.jar"), conf);
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_calib3d.so.2.3.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_calib3d.so.2.3#libopencv_calib3d.so
    .2.3"), conf);
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_contrib.so.2.3.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_contrib.so.2.3#libopencv_contrib.so.
    2.3"), conf);
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_core.so.2.3.0"), conf);
```

```
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_core.so.2.3#libopencv_core.so.2.3"),
conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_features2d.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_features2d.so.2.3#libopencv_feature
s2d.so.2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_flann.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_flann.so.2.3#libopencv_flann.so.2.3
"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_gpu.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_gpu.so.2.3#libopencv_gpu.so.2.3"),
conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_highgui.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_highgui.so.2.3#libopencv_highgui.s
o.2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_imgproc.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_imgproc.so.2.3#libopencv_imgproc.
so.2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_legacy.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_legacy.so.2.3#libopencv_legacy.so.
2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_ml.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_ml.so.2.3#libopencv_ml.so.2.3"),
conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_objdetect.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_objdetect.so.2.3#libopencv_objdetec
t.so.2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_ts.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_ts.so.2.3#libopencv_ts.so.2.3"),
conf);
```

```
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_video.so.2.3.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_video.so.2.3#libopencv_video.so.2.3
    "), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libtbb.so.2#libtbb.so.2"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavcodec.so.52#libavcodec.so.52"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavdevice.so.52#libavdevice.so.52"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavfilter.so.1#libavfilter.so.1"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavformat.so.52#libavformat.so.52"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavutil.so.50#libavutil.so.50"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libpostproc.so.51#libpostproc.so.51"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libswscale.so.0#libswscale.so.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libxvidcore.so.4#libxvidcore.so.4"), conf);


} catch (URISyntaxException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

    JobClient.runJob(conf);
    return 0;
}
public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new imageprocessor(), args);
    System.exit(exitCode);
}
}
```

SequenceFile İşleme Görevi Kodu:

```
import java.awt.Image;
import java.awt.Component;
import java.awt.Toolkit;
import java.awt.image.BufferedImage;
import java.awt.image.MemoryImageSource;
import java.awt.Rectangle;
```

```
import java.io.File;
import java.io.InputStream;
import java.net.URI;
import java.net.URISyntaxException;
import java.nio.ByteOrder;
import java.util.List;
import java.util.Scanner;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.imageio.stream.ImageInputStream;
import javax.imageio.stream.ImageOutputStream;
import javax.imageio.stream.MemoryCacheImageOutputStream;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.lib.IdentityReducer;
import org.apache.hadoop.mapred.lib.MultipleSequenceFileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


public class imageprocessor extends Configured implements Tool {

static class SequenceFileMapper extends MapReduceBase implements
Mapper<Text, BytesWritable, Text, BytesWritable> {
    private JobConf conf;
    @Override
    public void configure(JobConf conf) {
    this.conf = conf;
}
    @Override
```

```
    public void map(Text key, BytesWritable value,OutputCollector<Text,
    BytesWritable> output, Reporter reporter)throws IOException {

    String path = System.getProperty("java.library.path");
    System.setProperty("java.awt.headless","true");
    System.out.println(path);
    String filename = key.toString();
    byte[] bytes2 = value.getBytes();
    ImageInputStream in = ImageIO.createImageInputStream(new
    ByteArrayInputStream(bytes2));
    BufferedImage image_buffered = ImageIO.read(in);
    int imgWidth = image_buffered.getWidth();
    int imgHeight =image_buffered.getHeight();
    Rectangle[] faceRect;
    OpenCV opencv;
    Rectangle[] squares = new Rectangle[0];
    // OpenCV setup
    PImage img = new PImage(imgWidth, imgHeight);
    image_buffered.getRGB(0,0,imgWidth,imgHeight,img.pixels,0,imgWidth);
    opencv.allocate(img.width,img.height);
    opencv.copy(img);
    opencv.cascade("FRONTALFACE_ALT",true);
    faceRect = opencv.detect(true);

    for(int index = 0; index<faceRect.length;index++){
        BufferedImage faceImage = new BufferedImage (faceRect[index].width,
        faceRect[index].height,image_buffered.getType());
         faceImage.setRGB (0, 0, faceRect[index].width, faceRect[index].height,
        opencv.getBuffer().pixels, imgWidth *(faceRect[index].y-1) +
        faceRect[index].x , imgWidth);
         ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ImageIO.write(faceImage, "jpg", baos);
        BytesWritable imageDataOutput = new BytesWritable( baos.toByteArray());
        String imgName = filename.substring(0, filename.lastIndexOf("."));
        String imgFormatName = filename.substring (filename.lastIndexOf(".") + 1,
        filename.length());
        //String outputImgName = imgName+index+"."+imgFormatName;
        String outputImgName = imgName +index+ "("+faceRect[index].x+ ","+
        faceRect[index].y +")" + "."+imgFormatName;
        System.out.println(outputImgName);
        output.collect(new Text(outputImgName), imageDataOutput);
    }
}
}
@SuppressWarnings("deprecation")
@Override
public int run(String[] args) throws IOException {

    JobConf conf = new JobConf(imageprocessor.class);
```

```
    Scanner input = new Scanner(System.in);
    //JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (conf == null) {
    return -1;
}
String inputFolderName;
String outputFolderName;
if(args.length<2){
    inputFolderName = "SEQ1";
    outputFolderName = "face_01";
}else{
    inputFolderName = args[0];
    outputFolderName = args[1];
}
conf.setInputFormat(SequenceFileInputFormat.class);
conf.setOutputFormat(ImageFileOutputFormat.class);
conf.set("mapred.output.compress", "false");
conf.set("mapred.compress.map.output","false");
conf.set("mapred.max.split.size","16777216");
conf.set("mapred.map.tasks","20");
conf.setCompressMapOutput(false);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(BytesWritable.class);
conf.setMapperClass(SequenceFileMapper.class);
conf.setNumReduceTasks(0);
SequenceFileInputFormat.setInputPaths(conf, new Path(inputFolderName));
ImageFileOutputFormat.setOutputPath(conf, new Path(outputFolderName));
try {
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-android-arm.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-linux-x86.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-linux-x86_64.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-windows-x86.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-windows-x86_64.jar"), conf);
    DistributedCache.addFileToClassPath(new
    Path("/user/hadoop01/lib_opencv/javacv-macosx-x86_64.jar"), conf);
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_calib3d.so.2.3.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_calib3d.so.2.3#libopencv_calib3d.so
    .2.3"), conf);
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_contrib.so.2.3.0"), conf);
```

```
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_contrib.so.2.3#libopencv_contrib.so.
2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_core.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_core.so.2.3#libopencv_core.so.2.3"),
conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_features2d.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_features2d.so.2.3#libopencv_feature
s2d.so.2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_flann.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_flann.so.2.3#libopencv_flann.so.2.3
"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_gpu.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_gpu.so.2.3#libopencv_gpu.so.2.3"),
conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_highgui.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_highgui.so.2.3#libopencv_highgui.s
o.2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_imgproc.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_imgproc.so.2.3#libopencv_imgproc.
so.2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_legacy.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_legacy.so.2.3#libopencv_legacy.so.
2.3"), conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_ml.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_ml.so.2.3#libopencv_ml.so.2.3"),
conf);
//DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_objdetect.so.2.3.0"), conf);
DistributedCache.addCacheFile(new
URI("/user/hadoop01/lib_opencv/libopencv_objdetect.so.2.3#libopencv_objdetec
t.so.2.3"), conf);
```

```
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_ts.so.2.3.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_ts.so.2.3#libopencv_ts.so.2.3"),
    conf);
    //DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_video.so.2.3.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libopencv_video.so.2.3#libopencv_video.so.2.3
    "), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libtbb.so.2#libtbb.so.2"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavcodec.so.52#libavcodec.so.52"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavdevice.so.52#libavdevice.so.52"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavfilter.so.1#libavfilter.so.1"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavformat.so.52#libavformat.so.52"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libavutil.so.50#libavutil.so.50"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libpostproc.so.51#libpostproc.so.51"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libswscale.so.0#libswscale.so.0"), conf);
    DistributedCache.addCacheFile(new
    URI("/user/hadoop01/lib_opencv/libxvidcore.so.4#libxvidcore.so.4"), conf);

} catch (URISyntaxException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
JobClient.runJob(conf);
return 0;
}
public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new imageprocessor(), args);
    System.exit(exitCode);
}
}
```

KİŞİSEL YAYIN VE ESERLER

[1] **Demir İ.**, Sayar A., Dağıtık ve Paralel Görüntü İşleme İçin Hadoop eklentisi, *20. IEEE Sinyal İşleme ve İletişim Uygulamaları Kurultayı, SIU 2012*, Muğla, 18–20 Nisan 2012.

[2] **Demir İ.**, Sayar A., Hadoop Optimization For Massive Image Processing: Case Study Face Detection, *2nd World Conference On Innovation and Computer Sciences, INSODE 2012*, İzmir, 10–14 Mayıs 2012.

ÖZGEÇMİŞ

1980 yılında Van'da doğdu. İlköğrenimini Van'da, ortaokul ve lise öğrenimini Samsun'da tamamladı. 1998 yılında tam burslu olarak girdiği Bilkent Üniversitesi Bilgisayar Mühendisliği bölümünden 2003 yılında mezun oldu. 2003 yılından itibaren TÜBİTAK BİLGEM bünyesindeki Bilişim Teknolojileri Enstitüsü'nde Uzman Araştırmacı olarak çalışmaktadır. 2009 yılında başladığı Kocaeli Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'ndaki Yüksek Lisans eğitimine devam etmektedir.