Büyük Veri Analizi

Ders 1

Ali Mertcan KOSE Ph.D.
amertcankose@ticaret.edu.tr

İstanbul Ticaret Üniversitesi

İSTANBUL TİCARET
ÜNİVERSİTESİ

Giriş

Gerçek hayattaki veri kümeleri, tek bir makinenin belleğine sığabileceğinden kat kat daha büyük olabilir, bu veri kümelerini işlemek uzun sürebilir ve alışılmış yazılım araçları bu göreve uygun olmayabilir. Büyük verinin genel tanımı şudur: belleğe sığmayan veya yaygın yazılım yöntemleriyle makul bir sürede işlenemeyen veya analiz edilemeyen veri miktarı. Bazıları için büyük veri olan şey, diğerleri için büyük veri olmayabilir ve bu tanım kime sorduğunuza bağlı olarak değişebilir. Büyük Veri ayrıca 3 V ile de ilişkilendirilir (daha sonra 4 V'ye genişletildi):

Giriş

1. Hacim: Adından da anlaşılacağı gibi, büyük veri genellikle çok büyük veri hacimleriyle ilişkilendirilir. Neyin büyük olduğu bağlama bağlıdır: Bir sistem için gigabaytlarca veri büyük olabilirken, bir diğeri için petabaytlarca veriye ulaşmamız gerekir.
2. Çeşitlilik: Genellikle büyük veri, metin, video ve ses gibi farklı veri biçimleri ve türleriyle ilişkilendirilir. Veriler, ilişkisel tablolar gibi yapılandırılmış veya metin ve video gibi yapılandırılmamış olabilir.

Giriş

3. Hız: Verilerin oluşturulma ve depolanma hızı diğer sistemlerden daha hızlıdır ve daha sürekli üretilir. Akış verileri, telekomünikasyon operatörleri, çevrimiçi mağazalar ve hatta Twitter gibi platformlar tarafından üretilebilir.
4. Doğruluk: Bu daha sonra eklenmiştir ve kullanılan verileri ve anlamlarını bilmenin herhangi bir analiz çalışmasında önemli olduğunu göstermeye çalışır.Verilerin beklentilerimizle örtüşüp örtüşmediğini, dönüşüm sürecinin verileri değiştirmediğini ve toplanan verileri yansıtıp yansıtmadığını kontrol etmemiz gerekir.

Hadoop

Apache Hadoop, büyük hacimli verilerin paralel depolanması ve hesaplanması için oluşturulmuş bir yazılım bileşenleri kümesidir. Başlangıçtaki temel fikir, yaygın olarak bulunan bilgisayarları, arızalara karşı yüksek dirençli ve dağıtık hesaplamalı bir şekilde dağıtık bir şekilde kullanmaktı. Başarısıyla birlikte, Hadoop kümelerinde daha fazla üst düzey bilgisayar kullanılmaya başlandı, ancak standart donanımlar hala yaygın bir kullanım örneğidir. Paralel depolama derken, depolanan verileri bir ağ üzerinden birbirine bağlı birkaç düğüm kullanarak paralel bir şekilde depolayan ve geri alan herhangi bir sistemi kastediyoruz. Hadoop aşağıdakilerden oluşur:

Hadoop

Hadoop Common: temel ortak Hadoop öğeleri

Hadoop YARN: bir kaynak ve iş yöneticisi

Hadoop MapReduce: büyük ölçekli paralel işlem motoru

Hadoop Distributed File System (HDFS): Adından da anlaşılacağı gibi HDFS, yerel diskler kullanılarak birden fazla makineye dağıtılabilen ve büyük bir depolama havuzu oluşturulabilen bir dosya sistemidir:

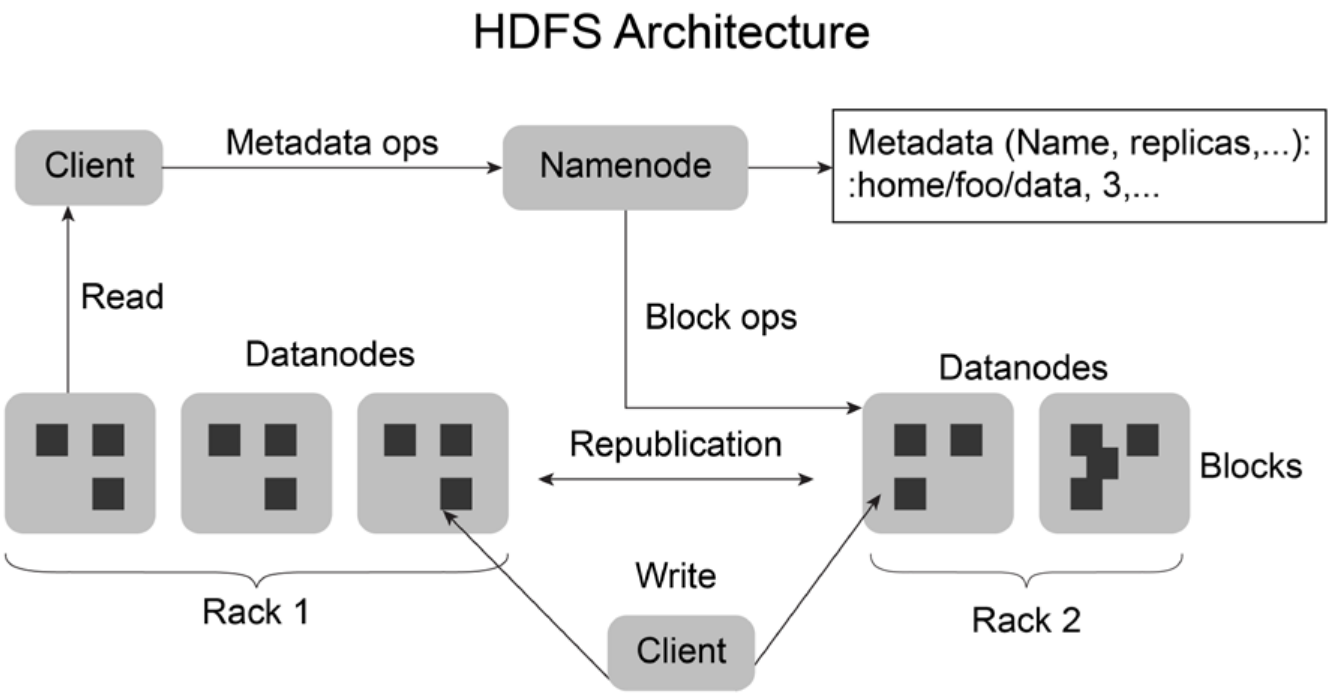

Figure 1: HDFS Mimarisi.

Hadoop

Bir diğer önemli bileşen ise Hadoop için bir kaynak yöneticisi ve iş planlayıcısı olan YARN'dır (Yet Another Resource Negotiator). Hadoop kümesine gönderilen işleri yönetmekten, gerekli ve mevcut kaynaklara göre bellek ve CPU tahsis etmekten sorumludur. Hadoop, ilk olarak Google tarafından geliştirilen dağıtılmış bir hesaplama paradigması olan MapReduce adlı paralel bir hesaplama modelini popüler hale getirmiştir. MapReduce kullanarak bir programı doğrudan Hadoop'ta çalıştırmak mümkündür. Ancak Hadoop'un ortaya çıkışından bu yana, Spark gibi başka paralel hesaplama paradigmaları ve çerçeveleri geliştirilmiştir, bu nedenle MapReduce veri analizi için yaygın olarak kullanılmaz. Spark'a dalmadan önce, HDFS'deki dosyaları nasıl işleyebileceğimizi görelim.

HDFS ile Verilerin İşlenmesi

HDFS, önemli bir ayrıcalığa sahip dağıtılmış bir dosya sistemidir: Özel olarak kendisi için üretilmemiş binlerce bilgisayarda (yani ticari donanım olarak adlandırılan) çalışmak üzere tasarlanmıştır. Herhangi bir özel ağ donanımı veya özel disk gerektirmez, sıradan donanımlarda çalışabilir. HDFS'yi tanımlayan bir diğer fikir de dayanıklı olmasıdır: Donanım her zaman arızalanacaktır, bu nedenle HDFS arızayı önlemeye çalışmak yerine, son derece hataya dayanıklı olarak bu sorunu aşar. Boyutu göz önüne alındığında arızaların meydana geleceğini varsayar, bu nedenle HDFS hızlı ve otomatik kurtarma için hata tespiti uygular. Ayrıca taşınabilirdir, çeşitli platformlarda çalışır ve terabaytlarca veri içeren tek dosyaları tutabilir.

HDFS ile Verilerin İşlenmesi

Kullanıcı açısından en büyük avantajlardan biri, HDFS'nin geleneksel hiyerarşik dosya yapısı organizasyonunu (klasörler ve dosyalar bir ağaç yapısı içinde) desteklemesidir. Böylece kullanıcılar her seviyede klasörler içinde klasörler ve klasörler içinde dosyalar oluşturabilir ve bu da kullanımını ve işleyişini basitleştirir. Dosyalar ve klasörler taşınabilir, silinebilir ve yeniden adlandırılabilir, böylece kullanıcıların HDFS'yi kullanmak için veri çoğaltma veya NameNode/DataNode mimarisi hakkında bilgi sahibi olmaları gerekmez; bir Linux dosya sistemine benzer. Dosyalara nasıl erişileceğini göstermeden önce, Hadoop verilerine erişmek için kullanılan adresler hakkında biraz açıklama yapmamız gerekir. Örneğin, HDFS'deki dosyalara erişim için URL şu biçimdedir: hdfs://hadoopnamenode.domainname/path/to/file Burada namenode.domainname, Hadoop'ta yapılandırılan adrestir. Hadoop kullanıcı kılavuzu (https://exitcondition.com/install-hadoop-windows/), Hadoop sisteminin farklı bölümlerine nasıl erişileceği hakkında biraz daha ayrıntılı bilgi verir. Tüm bunların nasıl çalıştığını daha iyi anlamak için birkaç örneğe bakalım.

Spark

Spark (https://spark.apache.org), büyük ölçekli veri işleme için birleşik bir analitik motorudur. Spark, 2009 yılında Kaliforniya Üniversitesi, Berkeley tarafından bir proje olarak başlatılmış ve 2013 yılında Apache Yazılım Vakfı'na taşınmıştır. Spark, veri akışı, HDFS'de depolanan dosyalar üzerinden SQL ve makine öğrenimi gibi analizler için kullanıldığında Hadoop mimarisiyle ilgili bazı sorunları çözmek üzere tasarlanmıştır. Verileri, bir kümedeki tüm işlem düğümlerine, her işlem adımının gecikmesini azaltacak şekilde dağıtabilir. Spark'ın bir diğer farkı da esnekliğidir: Java, Scala, SQL, R ve Python için arayüzler ve makine öğrenimi için MLlib, grafik hesaplama için GraphX ve akış iş yükleri için Spark Streaming gibi farklı sorunlar için kütüphaneler mevcuttur.

Spark

Spark, paralel yürütmeleri başlatmak için kullanıcı girdisini alan bir sürücü işlemine ve görevleri yürütmek için küme düğümlerinde bulunan çalışan işlemlerine sahip olan çalışan soyutlamasını kullanır. Dahili bir küme yönetim aracına sahiptir ve Hadoop YARN ve Apache Mesos (ve hatta Kubernetes) gibi diğer araçları destekleyerek farklı ortamlara ve kaynak dağıtım senaryolarına entegre olur. Spark ayrıca çok hızlı olabilir, çünkü öncelikle verileri tüm düğümlere dağıtmaya ve yalnızca diskteki verilere güvenmek yerine bellekte tutmaya çalışır. Toplam kullanılabilir bellekten daha büyük veri kümelerini işleyebilir, verileri bellek ve disk arasında kaydırabilir, ancak bu işlem, tüm veri kümesinin tüm düğümlerin toplam kullanılabilir belleğine sığması durumunda olduğundan daha yavaştır:

Spark

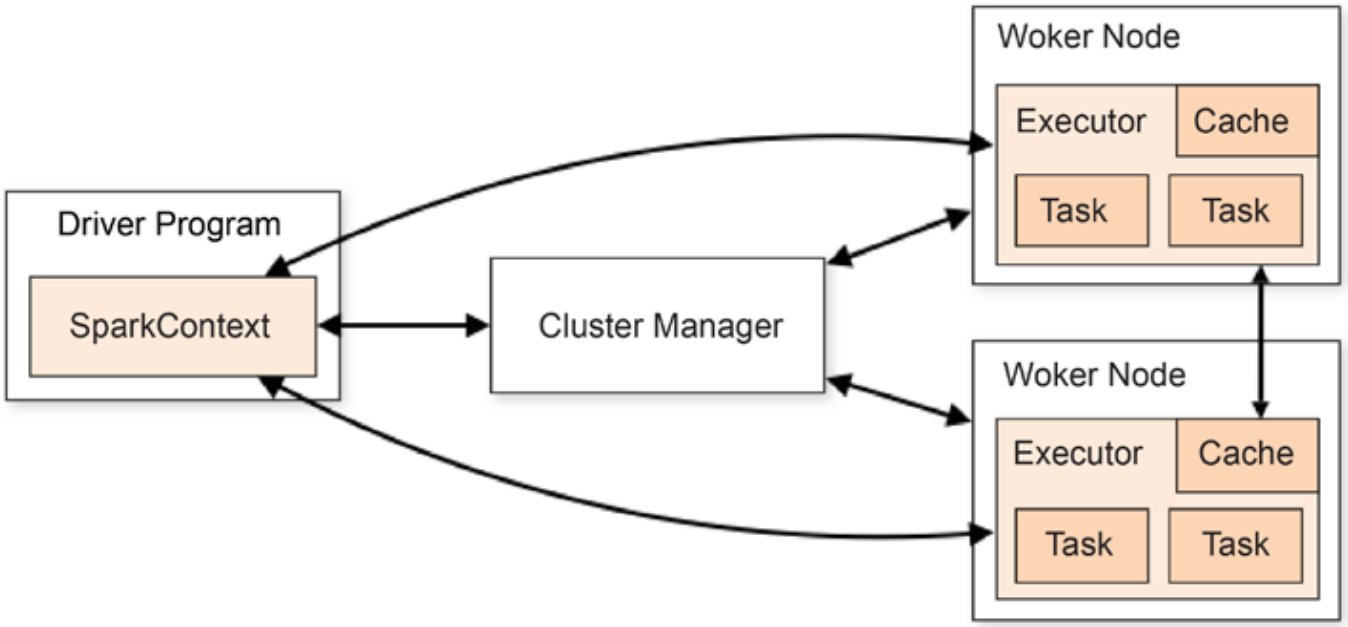

Figure 2: Spark Çalışma Mekanizması.

Spark

Spark'ın diğer büyük avantajları arasında, HDFS, Amazon S3, Cassandra ve diğerleri gibi çok çeşitli yerel ve dağıtık depolama sistemleri için arayüzlere sahip olması; JDBC veya ODBC bağlayıcıları aracılığıyla PostgreSQL ve MySQL gibi RDBMS'lere bağlanabilmesi; ve SQL'i doğrudan bir HDFS dosyası üzerinden çalıştırmak için Hive Metastore'u kullanabilmesi yer alır. CSV, Parquet ve ORC gibi dosya formatları da Spark tarafından doğrudan okunabilir. Bu esneklik, farklı formatlara sahip olabilen büyük veri kaynaklarıyla çalışırken büyük bir yardımcı olabilir. Spark, Scala, Python ve R ile etkileşimli bir kabuk olarak veya spark-submit komutuyla bir iş gönderme platformu olarak kullanılabilir. Gönderme yöntemi, işleri bir betikte kodlanmış bir Spark kümesine göndermek için kullanılır. Python için Spark kabuk arayüzüne PySpark denir. Varsayılan Python sürümünün kullanılacağı terminalden doğrudan erişilebilir; IPython kabuğunu veya hatta bir Jupyter not defterinin içinden erişilebilir.

Spark SQL and Pandas DataFrames

RDD veya Dayanıklı Dağıtılmış Veri Kümesi, Spark'ın verilerle çalışmak için kullandığı temel soyutlamadır. Spark 2.0 sürümünden itibaren, verileri işlemek için önerilen API DataFrame API'dir. DataFrame API, RDD API'nin üzerine kuruludur, ancak RDD API'ye hala erişilebilir. RDD'lerle çalışmak düşük seviyeli olarak kabul edilir ve tüm işlemler DataFrame API'de mevcuttur, ancak RDD API hakkında biraz daha fazla bilgi edinmekte fayda vardır. SQL modülü, kullanıcıların yaygın ilişkisel veritabanlarına benzer şekilde, Spark'taki verileri SQL sorguları kullanarak sorgulamasına olanak tanır. DataFrame API, yapılandırılmış verilerle çalışan SQL modülünün bir parçasıdır.

Spark SQL and Pandas DataFrames

Bu veri arayüzü, bu tür hesaplamaları ifade etmek için kullanılan API veya dilden bağımsız olarak, aynı yürütme motorunun kullanılmasıyla ek optimizasyonlar oluşturmaya yardımcı olur. DataFrame API, Pandas DataFrame'e benzer. Spark'ta bir DataFrame, her sütunun bir adı olan sütunlar halinde düzenlenmiş dağıtılmış bir veri koleksiyonudur. Spark 2.0 ile DataFrame, daha genel Veri Kümesi API'sinin bir parçasıdır, ancak bu API yalnızca Java ve Scala dilleri için mevcut olduğundan, yalnızca DataFrame API'sini (belgelerde Türsüz Veri Kümesi İşlemleri olarak adlandırılır) ele alacağız.

Spark SQL and Pandas DataFrames

Spark Veri Çerçeveleri arayüzü, Pandas arayüzüne benzer, ancak önemli farklılıklar vardır:

- İlk fark, Spark Veri Çerçevelerinin değiştirilemez olmasıdır: oluşturulduktan sonra değiştirilemezler.
- İkinci fark, Spark'ın iki farklı işlem türüne sahip olmasıdır: dönüşümler ve eylemler. Dönüşümler, bir Veri Çerçevesinin öğelerine uygulanan ve daha sonra yürütülmek üzere sıraya alınan, henüz veri almayan işlemlerdir. Yalnızca bir işlem çağrıldığında veri alınır ve sıradaki tüm dönüşümler yürütülür. Buna tembel değerlendirme denir.

Spark'ta DataFrame İşlemlerini Gerçekleştirme

Matplotlib ve Seaborn kullanılarak yapılan istatistiksel görselleştirmeler burada uygulanabilir ve bu da daha karmaşık analizleri daha hızlı gerçekleştirmeyi kolaylaştırır. Bu analizler, daha sonra toplanan veriler üzerinde toplama, istatistik, hesaplama ve görselleştirme gibi işlemleri içerir. Daha önce yaptığımız gibi, bir CSV dosyasını okuyarak üzerinde bazı analizler yapmak istiyoruz:

Spark'ta DataFrame İşlemlerini Gerçekleştirme

Öncelikle, Jupyter not defterinde bir Spark oturumu oluşturmak için aşağıdaki komutu kullanalım:

```
from pyspark.sql import SparkSession
»> spark = SparkSession
.builder  .appName("Python Spark Session")
.getOrCreate()n Spark Session")  \ .getOrCreate()}
```

Spark'ta DataFrame İşlemlerini Gerçekleştirme

```
df = spark.createDataFrame([
("Jonh", 22, 1.80), ( "Hughes", 34,1.96),
("Mary",27, 1.56)], ["name", "age", "height"])
df.show()
```

Spark'ta DataFrame İşlemlerini Gerçekleştirme

Veri Çerçevemizi okuyup içeriğini gösterdikten sonra, analiz yapabilmek için verileri düzenlemeye başlamak istiyoruz. Verilere, aynı NumPy seçim sözdizimini kullanarak, giriş olarak sütun adını vererek erişebiliriz. Döndürülen nesnenin Sütun türünde olduğunu unutmayın:

1. Önceki alıştırmada aldığımız DataFrame'den bir sütun seçelim:

```
df['age'].Column['age']
```

```
Bu, Pandas'ta gördüğümüzden farklı. Bir Spark DataFrame'in
sütunlardan değerleri seçen yöntem select'tir. Şimdi,
bu yöntemi kullandığımızda ne olacağını görelim.
```

Spark'ta DataFrame İşlemlerini Gerçekleştirme

2. Aynı Veri Çerçevesini tekrar kullanarak, select metodunu kullanarak isim sütununu seçin:

```
df.select(df['name'])DataFrame[age:  string]
```

Spark'ta DataFrame İşlemlerini Gerçekleştirme

3. Şimdi, Sütun'dan Veri Çerçevesi'ne geçtik. Bu sayede Veri Çerçeveleri için yöntemleri kullanabiliriz. Yaş için select yönteminden elde edilen sonuçları göstermek için show yöntemini kullanabilirsiniz:

```
df.select(df['age']).show()
```

Spark'ta DataFrame İşlemlerini Gerçekleştirme

4. Birden fazla sütun seçelim. Bunu yapmak için sütunların adlarını kullanabiliriz:

```
df.select(df['age'], df['height']).show()
```

Spark'ta DataFrame İşlemlerini Gerçekleştirme

Bu, aynı sözdizimini kullanarak, isme göre seçim yaparak diğer sütunlar için de genişletilebilir. Bir sonraki bölümde, GroupBy ile toplamalar gibi daha karmaşık işlemlere bakacağız.

Yerel Dosya Sisteminden ve HDFS'den Veri Okuma

Daha önce gördüğümüz gibi, yerel diskteki dosyaları okumak için Spark'a dosyanın yolunu vermeniz yeterlidir. Ayrıca, farklı depolama sistemlerinde bulunan diğer birçok dosya biçimini de okuyabiliriz. Spark, aşağıdaki biçimlerdeki dosyaları okuyabilir:

- CSV
- JSON
- ORC
- Parquet
- Text

Yerel Dosya Sisteminden ve HDFS'den Veri Okuma

Ve aşağıdaki depolama sistemlerinden okunabilir:

- JDBC
- ODBC
- Hive
- S3
- HDFS

Yerel Dosya Sisteminden ve HDFS'den Veri Okuma

Bir URL şemasına dayanarak, bir alıştırma olarak, farklı yerlerden ve formatlardan veri okuyalım:

1. Jupyter not defterine gerekli kütüphaneleri aktarın:

```
from pyspark.sql import SparkSession
spark = SparkSession
.builder
.appName("Python Spark Session")
.getOrCreate()
```

Yerel Dosya Sisteminden ve HDFS'den Veri Okuma

2. Web'deki API'lerden toplanan veriler için yaygın olan bir JSON dosyasından veri almamız gerektiğini varsayalım. Bir dosyayı doğrudan HDFS'den okumak için aşağıdaki URL'yi kullanın:

```
df = spark.read.json('hdfs://hadoopnamenode/data/myjsonfile.json')
```

```
Bu tür bir URL ile HDFS uç noktasının tam adresini sağlamamız gerektiğini
unutmayın.  Spark'ın doğru seçeneklerle yapılandırıldığını varsayarak,
yalnızca basitleştirilmiş yolu da kullanabiliriz.
```

Yerel Dosya Sisteminden ve HDFS'den Veri Okuma

3. Şimdi aşağıdaki komutu kullanarak verileri Spark nesnesine okuyun:

```
df = spark.read.json('hdfs://data/myjsonfile.json')
```

Yerel Dosya Sisteminden ve HDFS'den Veri Okuma

4. Okuma yönteminde formatı, erişim URL'sinde ise depolama sistemini seçiyoruz. JDBC bağlantılarına erişmek için de aynı yöntem kullanılır, ancak genellikle bağlanmak için bir kullanıcı adı ve parola girmemiz gerekir. Bir PostgreSQL veritabanına nasıl bağlanacağımıza bakalım:

```
{url = "jdbc:postgresql://posgreserver:5432/mydatabase"
\ properties = {"user":  "my_postgre_user", password:
"mypassword", \ "driver":  "org.postgresql.Driver"} \
df = spark.read.jdbc(url, table = "mytable", properties
= properties)]
```

Verileri HDFS ve PostgreSQL'e Geri Yazma

Pandas'ta gördüğümüz gibi, bazı işlemler ve dönüşümler gerçekleştirdikten sonra, sonuçları yerel dosya sistemine geri yazmak istediğimizi varsayalım. Bu, bir analizi tamamlayıp sonuçları diğer ekiplerle paylaşmak istediğimizde veya verilerimizi ve sonuçlarımızı başka araçlar kullanarak göstermek istediğimizde çok faydalı olabilir:

Verileri HDFS ve PostgreSQL'e Geri Yazma

1. Write metodunu doğrudan DataFrame'den HDFS'de kullanabiliriz:

```
df.write.csv('results.csv', header=True)
```

2. İlişkisel veritabanı için, burada gösterildiği gibi aynı URL'yi ve özellik sözlüğünü kullanın:

```
df = spark.write.jdbc(url, tablo = "mytable", özellikler
= özellikler)
```

Bu, Spark'a büyük veri kümelerini işleme ve bunları analiz için birleştirme konusunda büyük esneklik sağlar.

Parke Dosyaları Yazma

Parquet veri formatı (https://parquet.apache.org/), Hadoop ve Spark dahil olmak üzere farklı araçlar tarafından kullanılabilen ikili, sütunlu bir depolama biçimidir. Daha yüksek performans ve depolama kullanımı sağlamak için sıkıştırmayı destekleyecek şekilde tasarlanmıştır. Sütun odaklı tasarımı, verileri arayıp gereksiz satırlardaki değerleri atmak yerine yalnızca gerekli sütunlardaki veriler alındığından, performans için veri seçimine yardımcı olur. Bu da verilerin dağıtıldığı ve diskte bulunduğu büyük veri senaryolarında erişim süresini azaltır. Parquet dosyaları ayrıca harici uygulamalar tarafından, bir C++ kütüphanesi kullanılarak ve hatta doğrudan Pandas'tan okunup yazılabilir. Parquet kütüphanesi şu anda Arrow projesi (https://arrow.apache.org/) ile geliştirilmektedir. Spark'ta daha karmaşık sorgular düşünüldüğünde, verileri Parquet formatında depolamak, özellikle sorguların büyük bir veri kümesini araması gerektiğinde performansı artırabilir. Sıkıştırma, Spark'ta bir işlem yapılırken iletilmesi gereken veri hacmini azaltmaya yardımcı olarak ağ I/O'sini azaltır. Ayrıca şemaları ve iç içe geçmiş yapıları da destekler.

Parke Dosyaları Yazma

Spark'taki Parquet yazıcısının, mod (ekleme, üzerine yazma, yoksayma veya hata, varsayılan seçenek) ve sıkıştırma algoritmasını seçmek için bir parametre olan sıkıştırma gibi çeşitli seçenekleri vardır. Kullanılabilir algoritmalar şunlardır:

- gzip
- lzo
- brottli
- lz4
- Snappy
- Uncompressed

Varsayılan algoritma hızlıdır.

Parke Dosyaları Yazma

Diyelim ki çok sayıda CSV dosyası aldık ve bunlar üzerinde bazı analizler yapmamız gerekiyor. Ayrıca veri hacmini de azaltmamız gerekiyor. Bunu Spark ve Parquet kullanarak yapabiliriz.

Analizimize başlamadan önce CSV dosyalarını Parquet'e dönüştürelim:

1. Öncelikle HDFS'den CSV dosyalarını okuyalım:

```
df = spark.read.csv('hdfs:/data/very_large_file.csv', header=True)
```

Parke Dosyaları Yazma

2. DataFrame'deki CSV dosyalarını HDFS'ye geri yazın, ancak bu sefer Parquet formatında:

```
df.write.parquet('hdfs:/data/data_file', compression="snapp
```

3. Şimdi Parquet dosyasını yeni bir DataFrame'e okuyalım:

```
df_pq=spark.read.parquet("hdfs:/data/data_file")
```

Yapılandırılmamış Verilerin İşlenmesi

Yapılandırılmamış veriler genellikle sabit bir formatı olmayan verileri ifade eder. Örneğin, CSV dosyaları yapılandırılmıştır ve JSON dosyaları da yapılandırılmış olarak kabul edilebilir, ancak tablo biçiminde değildir. Diğer yandan, bilgisayar günlükleri aynı yapıya sahip değildir, çünkü farklı programlar ve arka plan programları ortak bir düzen olmadan mesajlar üretir. Resimler de serbest metin gibi yapılandırılmamış verilere bir başka örnektir.

Spark'ın veri okuma esnekliğinden yararlanarak yapılandırılmamış formatları ayrıştırabilir ve gerekli bilgileri daha yapılandırılmış bir formata dönüştürerek analize olanak sağlayabiliriz. Bu adıma genellikle ön işleme veya veri düzenleme denir.

Metni Ayrıştırma ve Temizleme

Bu alıştırmada, bir metin dosyasını okuyacağız, satırlara böleceğiz ve verilen dizeden the ve a kelimelerini kaldıracağız:

1. shake.txt metin dosyasını (https://raw.githubusercontent.com/TrainingByPackt/Big-Data-Analysis-with-Python/master/Lesson03/data/shake.txt) metin metodunu kullanarak Spark nesnesine okuyun:

```
from operator import add
rdd_df = spark.read.text("/shake.txt").rdd
```

Metni Ayrıştırma ve Temizleme

2. Aşağıdaki komutu kullanarak metinden satırları çıkarın:

```
lines = rdd_df.map(lambda line:  line[0])
```

3. Bu, dosyadaki her satırı listede bir giriş olarak böler. Sonucu kontrol etmek için, tüm verileri sürücü işlemine geri toplayan toplama yöntemini kullanabilirsiniz:

```
lines.collect()
```

4. Şimdi count metodunu kullanarak satır sayısını sayalım:

```
lines.count()
```

Metni Ayrıştırma ve Temizleme

5. Şimdi, önce her satırı kelimelere bölelim, etrafındaki boşluklara göre ayıralım ve tüm öğeleri birleştirip büyük harfli kelimeleri çıkaralım:

```
{splits = lines.flatMap(lambda x:  x.split(‘ ’)) \
lower_splits = splits.map(lambda x:  x.lower().strip())$}
```

6. Verilen dizeden the ve a kelimelerini ve ’.’, ’,’ gibi noktalama işaretlerini de kaldıralım:

```
prep = [‘the’, ‘a’, ‘,’, ‘.’]
```

Metni Ayrıştırma ve Temizleme

7. Token listemizden durdurma sözcüklerini kaldırmak için aşağıdaki komutu kullanın:

```
tokens = lowersplits.filter(lambda x:  x and x not in
prep)
```

Artık token listemizi işleyebilir ve benzersiz kelimeleri sayabiliriz. Buradaki fikir, ilk elemanın token, ikinci elemanın ise o token'ın sayısı olduğu bir tuple listesi oluşturmaktır.

8. Tokenımızı bir listeye eşleyelim:

```
token_list = tokens.map(lambda x:  [x, 1])
```

9. Her listeye işlemi uygulayacak olan reduceByKey işlemini kullanın:

```
count = token_list.reduceByKey(add).sortBy(lambda x:
x[1],
ascending=False)
```

Metni Ayrıştırma ve Temizleme

```
>>> from operator import add
>>> rdd_df = spark.read.text("shake.txt").rdd
>>> lines = rdd_df.map(lambda line: line[0])
>>> lines.collect()
['It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it wa
s the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it
 was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were
 all going direct to Heaven, we were all going direct the other way—in short, the period was so far like the present
period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlativ
e degree of comparison only.']
>>>
>>> splits = lines.flatMap(lambda x: x.split(' '))
>>> lower_splits = splits.map(lambda x: x.lower().strip())
>>> prep = ['the', 'a', ',', '.']
>>> tokens = lower_splits.filter(lambda x: x and x not in prep)
>>> token_list = tokens.map(lambda x: [x, 1])
>>> count = token_list.reduceByKey(add).sortBy(lambda x: x[1], ascending=False)
>>> count.collect()
[('of', 12), ('was', 11), ('it', 10), ('we', 4), ('times,', 2), ('age', 2), ('epoch', 2), ('season', 2), ('had', 2),
('before', 2), ('us,', 2), ('were', 2), ('all', 2), ('going', 2), ('direct', 2), ('its', 2), ('for', 2), ('best', 1),
 ('worst', 1), ('wisdom,', 1), ('foolishness,', 1), ('belief,', 1), ('incredulity,', 1), ('light,', 1), ('darkness,',
 1), ('spring', 1), ('hope,', 1), ('winter', 1), ('despair,', 1), ('everything', 1), ('nothing', 1), ('to', 1), ('hea
ven,', 1), ('other', 1), ('way—in', 1), ('short,', 1), ('period', 1), ('so', 1), ('far', 1), ('like', 1), ('present',
 1), ('period,', 1), ('that', 1), ('some', 1), ('noisiest', 1), ('authorities', 1), ('insisted', 1), ('on', 1), ('bei
ng', 1), ('received,', 1), ('good', 1), ('or', 1), ('evil,', 1), ('in', 1), ('superlative', 1), ('degree', 1), ('comp
arison', 1), ('only.', 1)]
>>>
```

Figure 3: Metni Ayrıştırma ve Temizleme.

Metinden Durdurma Sözcüklerini Kaldırma

Bu etkinlikte bir metin dosyasını okuyacağız, satırlara böleceğiz ve metinden durdurma kelimelerini kaldıracağız:

1. Yukarıdaka kullanılan shake.txt metin dosyasını okuyun.
2. Satırları metinden çıkarın ve her satırla bir liste oluşturun.
3. Her satırı, etrafındaki boşluklara bölerek kelimelere ayırın ve büyük harfli kelimeleri çıkarın.

Metinden Durdurma Sözcüklerini Kaldırma

4. Jeton listemizden “of”, “a”, “and”, “to” gibi durdurma kelimelerini kaldırın.
5. Jeton listesini işleyin ve benzersiz kelimeleri sayarak, jeton ve sayısından oluşan bir liste oluşturun.
6. Jetonlarımızı reduceByKey işlemini kullanarak bir listeye eşleyin.

Metinden Durdurma Sözcüklerini Kaldırma

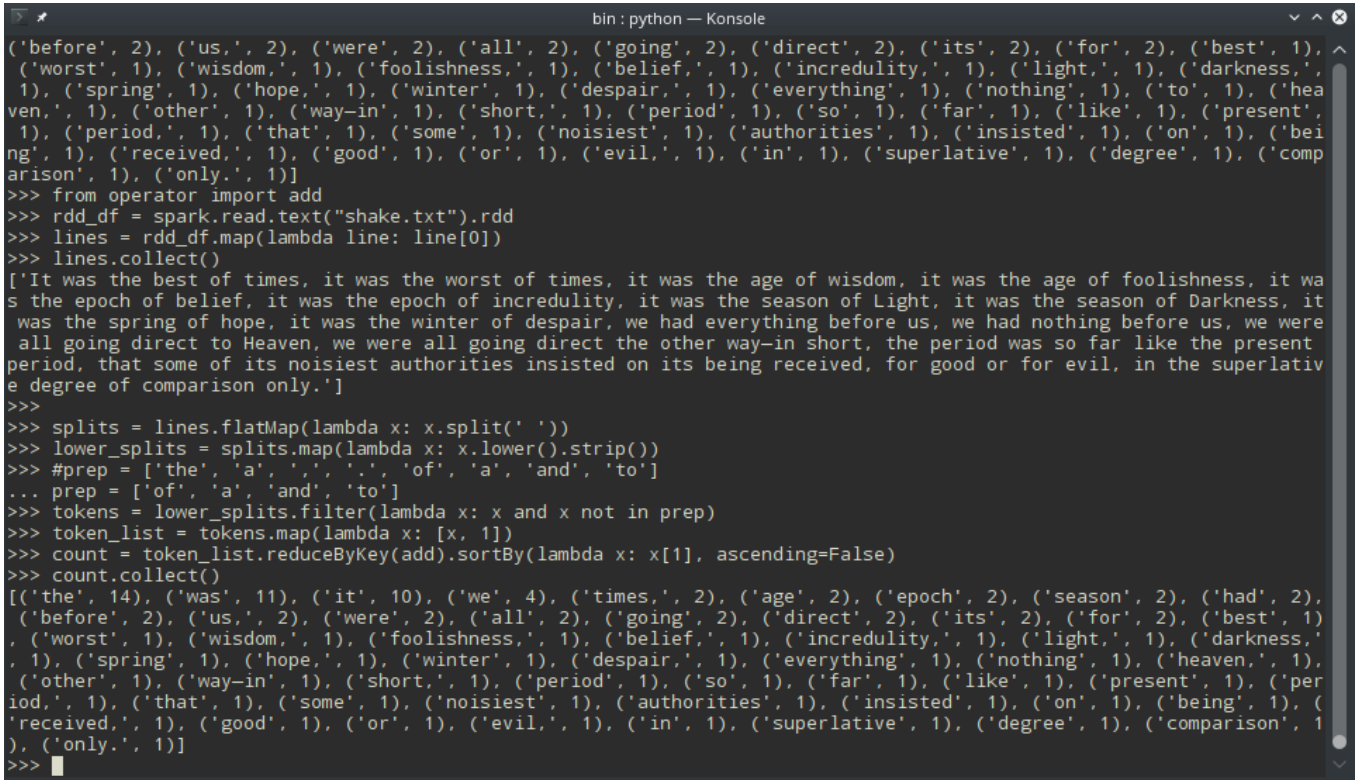

Figure 4: Metinden Durdurma Sözcüklerini Kaldırma.

Metinden Durdurma Sözcüklerini Kaldırma

Her bir tuple'ın bir belirteç ve o kelimenin metinde kaç kez geçtiğini gösteren tuple'ların listesini elde ederiz. Son toplama sayısından (bir eylem) önce, dönüşüm olan işlemlerin hemen çalışmaya başlamadığına dikkat edin: Spark'ın tüm adımları yürütmeye başlaması için "işlem sayısı" eylemine ihtiyacımız vardı. Diğer yapılandırılmamış veri türleri, önceki örnek kullanılarak ayrıştırılabilir ve önceki etkinlikte olduğu gibi doğrudan üzerinde işlem yapılabilir veya daha sonra bir Veri Çerçevesine dönüştürülebilir.