Big Data Analysis with Apache Spark
This Lecture

Resilient Distributed Datasets (RDDs)

Creating an RDD

Spark RDD Transformations and Actions

Spark RDD Programming Model

Spark Shared Variables
Review: Python Spark (pySpark)

We are using the Python programming interface to Spark (pySpark)

pySpark provides an easy-to-use programming abstraction and parallel runtime:

» “Here’s an operation, run it on all of the data”

DataFrames are the key concept
A Spark program is two programs:

» A **driver program** and a **workers program**

Worker programs run on cluster nodes or in local threads.

DataFrames are distributed across workers.
Review: Spark and SQL Contexts

A Spark program first creates a **SparkContext** object

» **SparkContext** tells Spark how and where to access a cluster

» pySpark shell, Databricks CE automatically create **SparkContext**

» **iPython** and programs must create a new **SparkContext**

The program next creates a **sqlContext** object

Use **sqlContext** to create DataFrames

In the labs, we create the SparkContext and sqlContext for you
Review: DataFrames

The primary abstraction in Spark

» **Immutable once constructed**
» Track lineage information to efficiently recompute lost data
» Enable operations on collection of elements in parallel

You construct DataFrames

» by *parallelizing* existing Python collections (lists)
» by *transforming* an existing Spark or pandas DFs
» from *files* in HDFS or any other storage system
Review: DataFrames

Two types of operations: transformations and actions

Transformations are lazy (not computed immediately)

Transformed DF is executed when action runs on it

Persist (cache) DFs in memory or disk
Resilient Distributed Datasets

*Untyped* Spark abstraction underneath DataFrames:

- Immutable once constructed
- Track lineage information to efficiently recompute lost data
- Enable operations on collection of elements in parallel

You construct RDDs

- by *parallelizing* existing Python collections (lists)
- by *transforming* an existing RDDs or DataFrame
- from *files* in HDFS or any other storage system

http://spark.apache.org/docs/latest/api/python/pyspark.html
Programmer specifies number of partitions for an RDD

(Default value used if unspecified)

**RDDs**

**RDD split into 5 partitions**

```
item-1
item-2
item-3
item-4
item-5
```
```
item-6
item-7
item-8
item-9
item-10
```
```
item-11
item-12
item-13
item-14
item-15
```
```
item-16
item-17
item-18
item-19
item-20
```
```
item-21
item-22
item-23
item-24
item-25
```

**more partitions = more parallelism**
RDDS

Two types of operations: \textit{transformations} and \textit{actions}

Transformations are lazy (\textit{not computed immediately})

Transformed RDD is executed when action runs on it

Persist (cache) RDDs in memory or disk
When to Use DataFrames?

Need high-level transformations and actions, and want high-level control over your dataset

Have typed (structured or semi-structured) data

You want DataFrame optimization and performance benefits
  » Catalyst Optimization Engine
    • 75% reduction in execution time
  » Project Tungsten off-heap memory management
    • 75+% reduction in memory usage (less GC)
DataFrame Performance (I)

Faster than RDDs

Benefits from Catalyst optimizer
DataFrame Performance (II)

Benefits from Project Tungsten

DataFrames vs. RDDs

Memory Usage when Caching

Data Size (GB)

0 15 30 45 60

Data set size (relative)

1x 2x 4x 8x 16x

Data frames

- Default
- Code Gen
- Tungsten onheap
- Tungsten offheap
When to Use RDDs?

Need low-level transformations and actions, and want low-level control over your dataset

Have unstructured or schema-less data (e.g., media or text streams)

Want to manipulate your data with functional programming constructs other than domain specific expressions

You don’t want the optimization and performance benefits available with DataFrames
Working with RDDs

Create an RDD from a data source:

Apply transformations to an RDD: map, filter

Apply actions to an RDD: collect, count

**collect** action causes parallelize, filter, and map transforms to be executed
Creating an RDD

Create RDDs from Python collections (lists)

```python
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> rDD = sc.parallelize(data, 4)
>>> rDD
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

No computation occurs with `sc.parallelize()`
- Spark only records how to create the RDD with four partitions
Creating RDDs

From HDFS, text files, [Hypertable](#), [Amazon S3](#), [Apache Hbase](#), SequenceFiles, any other Hadoop [InputFormat](#), and directory or glob wildcard: `/data/201404*`

```python
>>> distFile = sc.textFile("README.md", 4)
```

```python
>>> distFile

MappedRDD[2] at textFile at

NativeMethodAccessorImpl.java:-2
```
Creating an RDD from a File

\[ \text{distFile} = \text{sc.textFile}("...", 4) \]

RDD distributed in 4 partitions

Elements are lines of input

*Lazy evaluation* means no execution happens now
Spark Transformations

Create new datasets from an existing one

Use \textit{lazy evaluation}: results not computed right away – instead Spark remembers set of transformations applied to base dataset

\begin{itemize}
  \item Spark optimizes the required calculations
  \item Spark recovers from failures and slow workers
\end{itemize}

Think of this as a recipe for creating result
### Some Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(func)</td>
<td>return a new distributed dataset formed by passing each element of the source through a function <code>func</code></td>
</tr>
<tr>
<td>filter(func)</td>
<td>return a new dataset formed by selecting those elements of the source on which <code>func</code> returns true</td>
</tr>
<tr>
<td>distinct([numTasks]))</td>
<td>return a new dataset that contains the distinct elements of the source dataset</td>
</tr>
<tr>
<td>flatMap(func)</td>
<td>similar to map, but each input item can be mapped to 0 or more output items (so <code>func</code> should return a <code>Seq</code> rather than a single item)</td>
</tr>
</tbody>
</table>
Transformations

```python
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]

>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]

>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

Function literals (green) are closures automatically passed to workers.
Transformations

```python
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

Function literals (green) are closures automatically passed to workers.
Transforming an RDD

```
lines = sc.textFile("...", 4)

comments = lines.filter(isComment)
```

Lazy evaluation means nothing executes – Spark saves recipe for transforming source
Spark Actions

Cause Spark to execute recipe to transform source

Mechanism for getting results out of Spark
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduce(func)</code></td>
<td>aggregate dataset's elements using function <code>func</code>. <code>func</code> takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel</td>
</tr>
<tr>
<td><code>take(n)</code></td>
<td>return an array with the first <code>n</code> elements</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>return all the elements as an array</td>
</tr>
<tr>
<td><code>takeOrdered(n, key=func)</code></td>
<td>return <code>n</code> elements ordered in ascending order or as specified by the optional key function</td>
</tr>
</tbody>
</table>
Getting Data Out of RDDs

```python
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
```
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] # as list
Spark Key-Value RDDs

Similar to Map Reduce, Spark supports Key-Value pairs

Each element of a Pair RDD is a pair tuple

```python
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```
## Some Key-Value Transformations

<table>
<thead>
<tr>
<th>Key-Value Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduceByKey(func)</code></td>
<td>return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function <code>func</code>, which must be of type <code>(V,V) ➔ V</code></td>
</tr>
<tr>
<td><code>sortByKey()</code></td>
<td>return a new dataset (K,V) pairs sorted by keys in ascending order</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>return a new dataset of (K, Iterable&lt;V&gt;) pairs</td>
</tr>
</tbody>
</table>
Key-Value Transformations

```python
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]

>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] → [(1,'a'), (1,'b'), (2,'c')]
```
Key-Value Transformations

```python
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] → [(1,['a','b']), (2,['c'])]
```

Be careful using `groupByKey()` as it can cause a lot of data movement across the network and create large Iterables at workers
Spark Programming Model

lines = sc.textFile("...", 4)

print lines.count()

count() causes Spark to:
- read data
- sum within partitions
- combine sums in driver
Spark Programming Model

```python
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

Spark recomputes `lines`:
- read data (again)
- sum within partitions
- combine sums in driver
Caching RDDs

```python
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

![Diagram showing caching and RAM usage]
Spark Program Lifecycle with RDDs

1. Create RDDs from external data or parallelize a collection in your driver program
2. Lazily transform them into new RDDs
3. cache() some RDDs for reuse
4. Perform actions to execute parallel computation and produce results
pySpark Closures

Spark automatically creates closures for:

» Functions that run on RDDs at executors
» Any global variables used by those executors

One closure per executor

» Sent for every task
» No communication between executors
» Changes to global variables at executors are not sent to driver
Consider These Use Cases

Iterative or single jobs with large global variables
  » Sending large read-only lookup table to executors
  » Sending large feature vector in a ML algorithm to executors

Counting events that occur during job execution
  » How many input lines were blank?
  » How many input records were corrupt?
Consider These Use Cases

Iterative or single jobs with large global variables
» Sending large read-only lookup table to executors
» Sending large feature vector in a ML algorithm to executors

Counting events that occur during job execution
» How many input lines were blank?
» How many input records were corrupt?

Problems:
• Closures are (re-)sent with every job
• Inefficient to send large data to each worker
• Closures are one way: driver → worker
Broadcast Variables

- Efficiently send large, *read-only* value to all executors
- Saved at workers for use in one or more Spark operations
- Like sending a large, read-only lookup table to all the nodes

Accumulators

- Aggregate values from executors back to driver
- Only driver can access value of accumulator
- For tasks, accumulators are write-only
- Use to count errors seen in RDD across executors
Broadcast Variables

Keep *read-only* variable cached on executors
  » Ship to each worker only once instead of with each task

Example: efficiently give every executor a large dataset

Usually distributed using efficient broadcast algorithms

At the driver:

```python
>>> broadcastVar = sc.broadcast([1, 2, 3])
```

At an executor (in code passed via a closure)

```python
>>> broadcastVar.value
[1, 2, 3]
```
Broadcast Variables Example

Country code lookup for HAM radio call signs

# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x+ y)))

Expensive to send large table
(Re-)sent for every processed file

From: http://shop.oreilly.com/product/0636920028512.do
Broadcast Variables Example

Country code lookup for HAM radio call signs

```python
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x + y)))
```

From: http://shop.oreilly.com/product/0636920028512.do
Accumulators

Variables that can only be “added” to by associative op

Used to efficiently implement parallel counters and sums

Only driver can read an accumulator’s value, not tasks

```python
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
...     global accum
...     accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```
Accumulators Example

Counting empty lines

```python
def extractCallSigns(line):
    global blankLines  # Make the global variable accessible
    if (line != ''):
        blankLines += 1
    return line.split(' ')

callSigns = fileflatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```

```python
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)
```
Accumulators

Tasks at executors cannot access accumulator’s values

Tasks see accumulators as write-only variables

Accumulators can be used in actions or transformations:
» Actions: each task’s update to accumulator is applied only once
» Transformations: no guarantees (use only for debugging)

Types: integers, double, long, float
» See lab for example of custom type
Summary

Master parameter specifies number of executors

Spark automatically pushes closures to Spark executors at workers

Programmer specifies number of partitions

Worker
- code
- RDD

Worker
- code
- RDD

Worker
- code
- RDD
Review: The Big Picture

Files: This lecture

Application Database

Extract Transform Load

Data Warehouse

Data Products

Business Intelligence

Analytics
What is a File?

A **file** is a named sequence of **bytes**
» Typically stored as a collection of pages (or blocks)

A **filesystem** is a collection of files organized within a hierarchical namespace
» Responsible for laying out those bytes on physical media
» Stores file metadata
» Provides an API for interaction with files

Standard operations
- open() / close()
- seek()
- read() / write()
Files: Hierarchical Namespace

On Mac and Linux, / is the root of a filesystem

On Windows, \ is the root of a filesystem

Files and directories have associated permissions

Files are not always arranged in a hierarchically
   » Content-addressable storage (CAS)
      » Often used for large multimedia collections
Considerations for a File Format

Data model: tabular, hierarchical, array

Physical layout

Field units and validation

Metadata: header, side file, specification, other?

Plain text (ASCII, UTF-8, other) or binary

Delimiters and escaping

Compression, encryption, checksums?

Schema evolution

Performance!
File Performance Considerations

Read versus write performance

Plain text versus binary format

Environment: Pandas (Python) versus Scala/Java

Uncompressed versus compressed
## File Performance

<table>
<thead>
<tr>
<th></th>
<th>Read Time (Text)</th>
<th>Write Time (Text)</th>
<th>Read Time (Binary)</th>
<th>Write Time (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pandas (Python)</td>
<td>36 secs</td>
<td>45 secs</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>Scala/Java</td>
<td>18 secs</td>
<td>21 secs</td>
<td>1-6* secs</td>
<td>1-6* secs</td>
</tr>
</tbody>
</table>

** Read-Write Times Comparable

** Pandas doesn’t have a binary file I/O library  
(Python performance depends on library you use)

* 6 seconds is the time for sustained read/write  
(often faster due to system caching)
## File Performance

<table>
<thead>
<tr>
<th></th>
<th>Read Time (Text)</th>
<th>Write Time (Text)</th>
<th>Read Time (Binary)</th>
<th>Write Time (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pandas (Python)</td>
<td>36 secs</td>
<td>45 secs</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>Scala/Java</td>
<td>18 secs</td>
<td>21 secs</td>
<td>1-6* secs</td>
<td>1-6* secs</td>
</tr>
</tbody>
</table>

- **Binary I/O much faster than text**

** Pandas doesn’t have a binary file I/O library
  (Python performance depends on library you use)

* 6 seconds is the time for sustained read/write
  (often faster due to system caching)

---

626 MB text file
787 MB binary file
## File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (Java default)</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td>787 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (default)</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

*Write times much larger than read*

*Scala/Java language*
### File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (Java default)</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td>787 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (default)</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

*Scala/Java language*

Large range of compression times
## File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (Java default)</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td>787 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (default)</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

**Scala/Java language**

Large range of compression times
## File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (Java default)</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td>787 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (default)</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

Small range (15%) of compressed file sizes

Scala/Java language
## File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (Java default)</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td>787 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (default)</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

Binary I/O still much faster than text.

Scala/Java language
## File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (Java default)</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (default)</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

*Scala/Java language*

Binary I/O still much faster than text
## File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (Java default)</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td>787 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6 (default)</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

Scala/Java language

LZ4 compression ≈ raw I/O speed
## File Performance - Compression

<table>
<thead>
<tr>
<th>Binary File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6</td>
<td>4 secs</td>
<td>75 secs</td>
<td>286 MB</td>
</tr>
<tr>
<td>(Java default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>4 secs</td>
<td>20 secs</td>
<td>313 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>4 secs</td>
<td>14 secs</td>
<td>328 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>2 secs</td>
<td>4 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw binary file</td>
<td>1-6 secs</td>
<td>1-6 secs</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text File</th>
<th>Read Time</th>
<th>Write Time</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip level 6</td>
<td>26 secs</td>
<td>98 secs</td>
<td>243 MB</td>
</tr>
<tr>
<td>(default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gzip level 3</td>
<td>25 secs</td>
<td>46 secs</td>
<td>259 MB</td>
</tr>
<tr>
<td>Gzip level 1</td>
<td>25 secs</td>
<td>33 secs</td>
<td>281 MB</td>
</tr>
<tr>
<td>LZ4 fast</td>
<td>22 secs</td>
<td>24 secs</td>
<td>423 MB</td>
</tr>
<tr>
<td>Raw text file</td>
<td>18 secs</td>
<td>21 secs</td>
<td>626 MB</td>
</tr>
</tbody>
</table>

*Scala/Java language*

LZ4 compression ≈ raw I/O speed
File Performance - Summary

• Uncompressed read and write times are comparable

• Binary I/O is much faster than text I/O

• Compressed reads much faster than compressed writes
  » LZ4 is better than gzip
  » LZ4 compression times approach raw I/O times
Lab: Text Analysis and Entity Resolution

Entity Resolution (ER) or Record linkage:
» Common, yet difficult problem in data cleaning and integration
» ER is term used by statisticians, epidemiologists, and historians
» Describes process of joining records from one data source with those from another dataset that describe same entity (e.g., data files, books, websites, databases)

A dataset that has undergone ER is referred to as being cross-linked
Entity Resolution

Use ER when joining datasets with entities that do not share a common identifier (e.g., DB key, URI, ID number)

Also, when they do share common ID, but have differences, such as record shape, storage location, and/or curator style
Entity Resolution Lab
Web scrape of Google Shopping and Amazon product listings

Google listing:
» clickart 950000 - premier image pack (dvd-rom) massive collection of images & fonts for all your design needs on dvd-rom! product information inspire your creativity and perfect any creative project with thousands of world-class images in virtually every style. plus clickart 950000 makes it easy for …

Amazon listing:
» clickart 950000 - premier image pack (dvd-rom)

Visually, we see these listings are the same product

How to algorithmically decide?

https://code.google.com/p/metric-learning/
Model and Algorithm

Model ER as *Text Similarity*

» We will use a weighted bag-of-words comparison
  • Some tokens (words) are more important than others
  • Sum up the weights of tokens in each document
Model and Algorithm

Model ER as *Text Similarity*

» We will use a weighted bag-of-words comparison
  • Some tokens (words) are more important than others
  • Sum up the weights of tokens in each document

How to assign weights to tokens?

» *Term-Frequency/Inverse-Document-Frequency* or *TF-IDF* for short
TF-IDF

**Term-Frequency**

« Rewards tokens that appear many times in the same document

\[ \text{score} = \frac{\text{# times } token \text{ appears}}{\text{total # of } tokens \text{ in } doc} \]
TF-IDF

**Term-Frequency**
- Rewards tokens that appear many times in the same document
  \[ = \frac{\text{(# times } \text{token appears})}{\text{(total # of tokens in doc)}} \]

**Inverse-Document-Frequency**
- Rewards tokens that are rare overall in a dataset
  \[ = \frac{\text{(total # of docs)}}{\text{(# of docs containing token)}} \]
TF-IDF

**Term-Frequency**
» Rewards tokens that appear many times in the same document
  = (# times *token* appears)/(total # of *tokens* in *doc*)

**Inverse-Document-Frequency**
» rewards tokens that are rare overall in a dataset
  = (total # of *docs*)/(# of *docs* containing *token*)

**Total TF-IDF value for a token**
» Product of its TF and IDF values for each doc
» Need to remove *stopwords* (a, the, …) from docs
Token Vectors

Formal method for text similarity (string distance metric):
» Treat each document as a vector in high dimensional space

Each unique token is a dimension
» Token weights are magnitudes in their respective token dimensions

Simple count-based vector example
» Document: “Hello, world! Goodbye, world!”
» Vector:

<table>
<thead>
<tr>
<th>Token</th>
<th>hello</th>
<th>goodbye</th>
<th>world</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Cosine Similarity

Compare two docs by computing cosine of angle between vectors

Small angle (large cosine) means docs share many tokens in common

Large angle (small cosine) means they have few words in common