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)

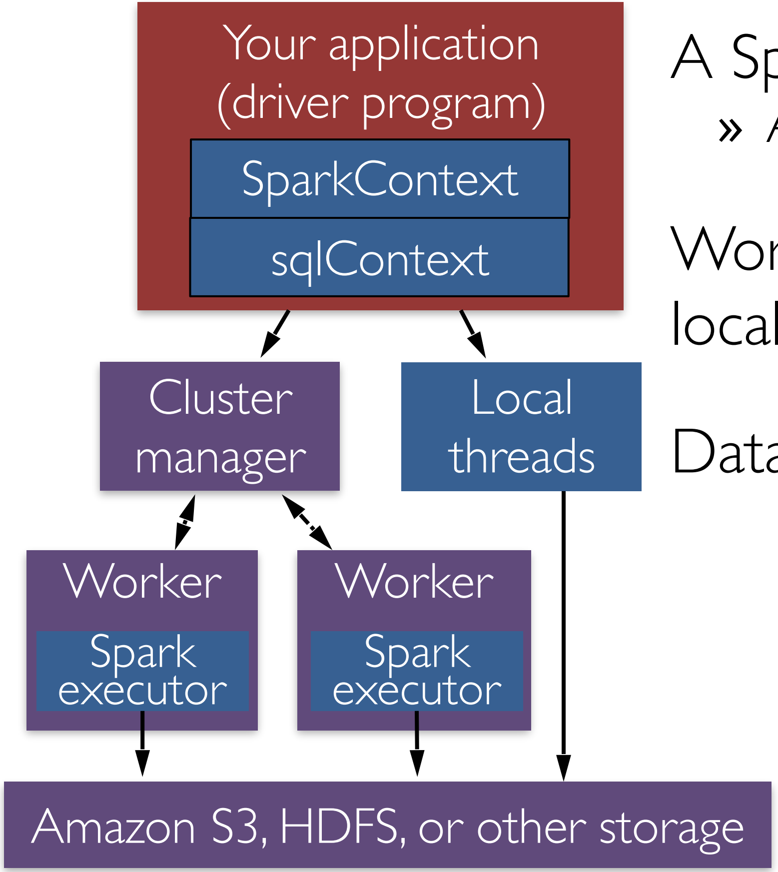
We are using the Python programming interface to Spark (pySpark)

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

Review: Spark Driver and Workers



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

[redacted]

http://spark.apache.org/docs/latest/api/python/pyspark.html

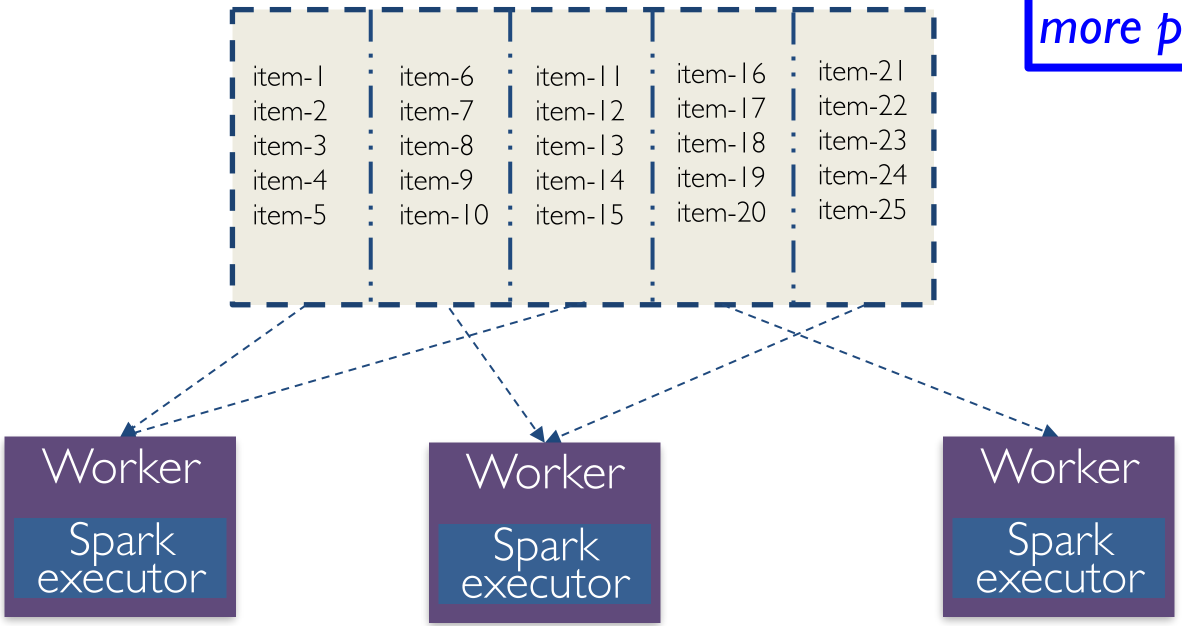
RDDs

Programmer specifies number of partitions for an RDD

(Default value used if unspecified)

RDD split into 5 partitions

more partitions = more parallelism



RDDs

Two types of operations: *transformations* and *actions*

Transformations are lazy (*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

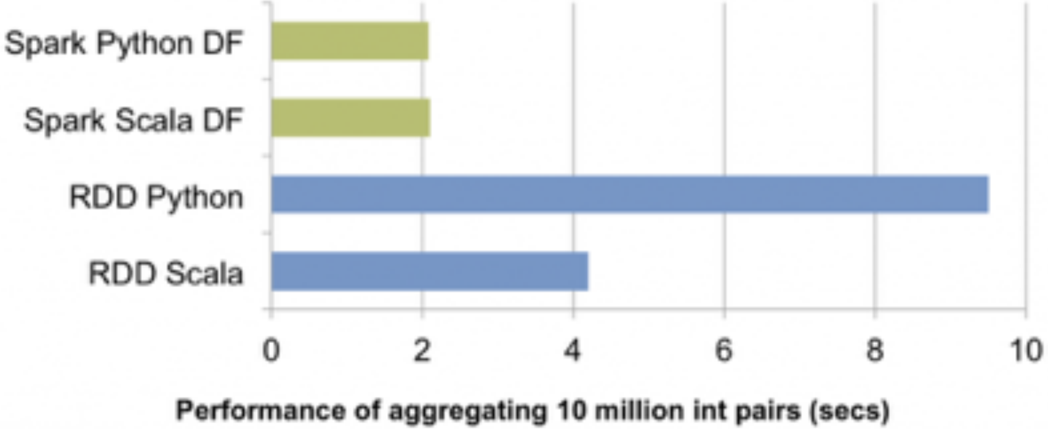
Have typed (structured or semi-structured) data

You want DataFrame optimization and performance benefits

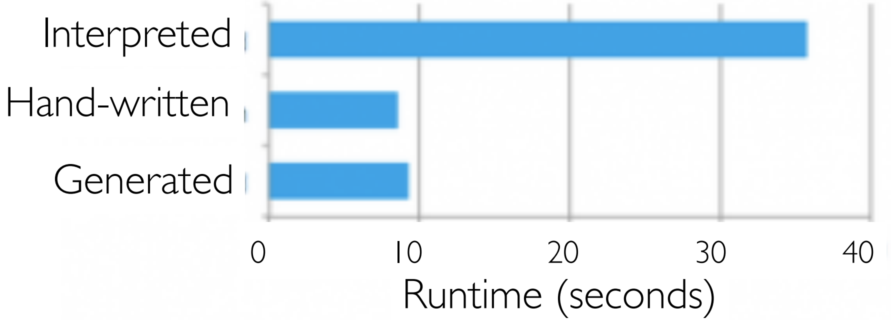
- » Catalyst Optimization Engine
  - 75% reduction in execution time
- » Project Tungsten off-heap memory management
  - 75+% reduction in memory usage (less GC)

DataFrame Performance (I)

Faster than RDDs

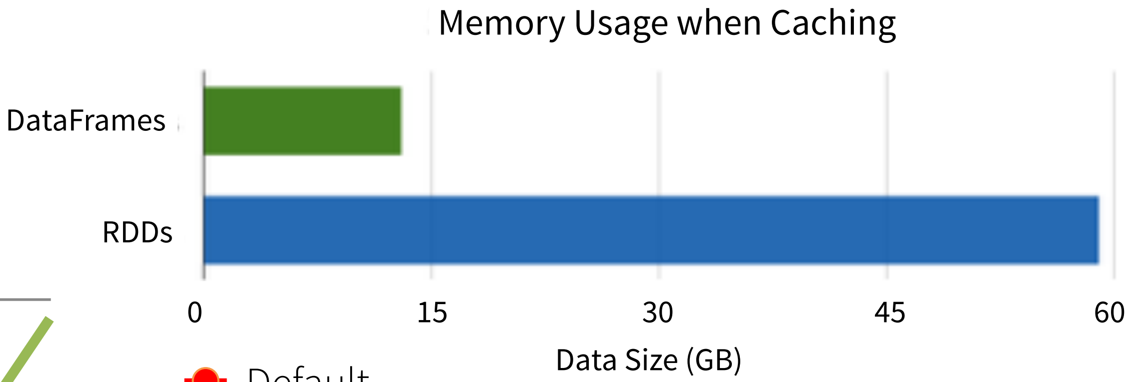


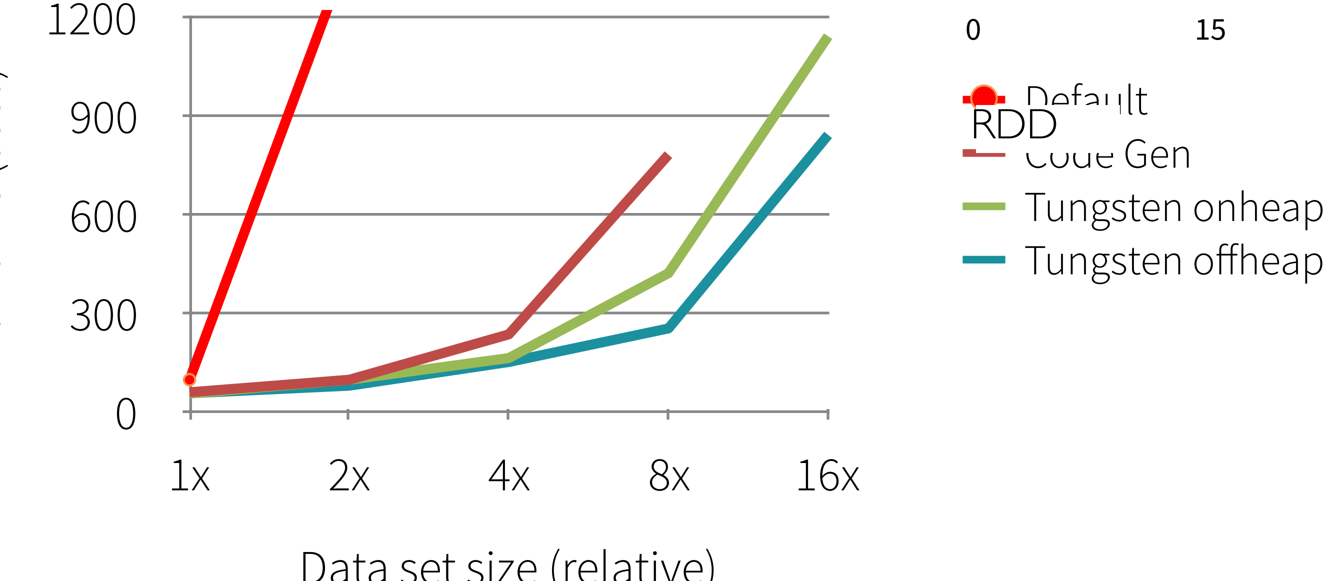
Benefits from Catalyst optimizer



DataFrame Performance (II)

Benefits from Project Tungsten





When to Use RDDs?

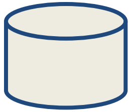
Need low-level transformations and actions, and want low-level control over your dataset

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

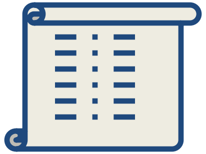
You don't want the optimization and performance benefits available with DataFrames

Working with RDDs

Create an RDD from a data source:  <list>

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)

No computation occurs with **`sc.parallelize()`**

- Spark only records how to create the RDD with four partitions

```
>>> 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
```

[redacted]

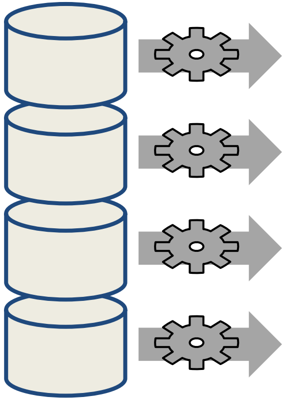
Creating RDDs

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

```
>>> distFile = sc.textFile("README.md", 4)
>>> distFile
MappedRDD[2] at textFile at
    NativeMethodAccessorImpl.java:-2
```

Creating an RDD from a File

```
distFile = 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 ***lazy evaluation***: results not computed right away – instead Spark remembers set of transformations applied to base dataset

» Spark optimizes the required calculations

» Spark recovers from failures and slow workers

Think of this as a recipe for creating result

Some Transformations

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

[redacted]

Transformations

```
>>> 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

[redacted]

Transformations

```
>>> 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]
```

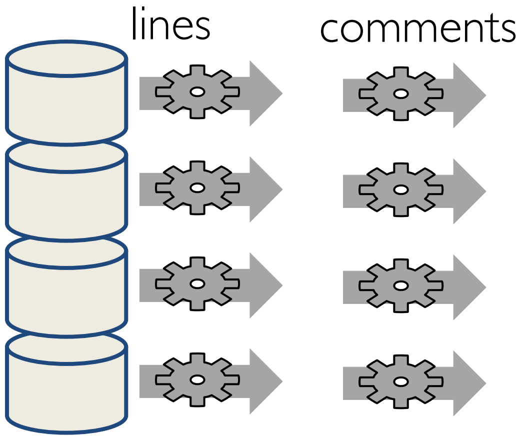
Function literals (green) are closures automatically passed to workers

[redacted]

Transforming an RDD

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

comments = lines.filter(isComment)
```



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

[redacted]

Spark Actions

Cause Spark to execute recipe to transform source

Mechanism for getting results out of Spark

Some Actions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

Getting Data Out of RDDs

```
>>> 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
```

Getting Data Out of RDDs

```
>>> 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

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

Some Key-Value Transformations

Key-Value Transformation	Description
`reduceByKey(`*func*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Key-Value Transformations

```
>>> 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

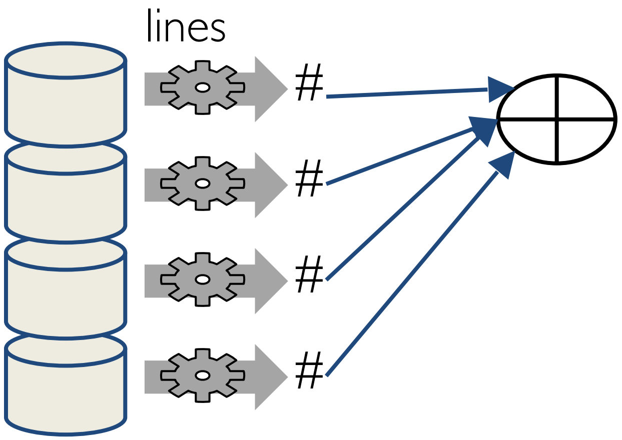
```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] →
              [(1,['a','b']), (2,['c'])]
```

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()
```

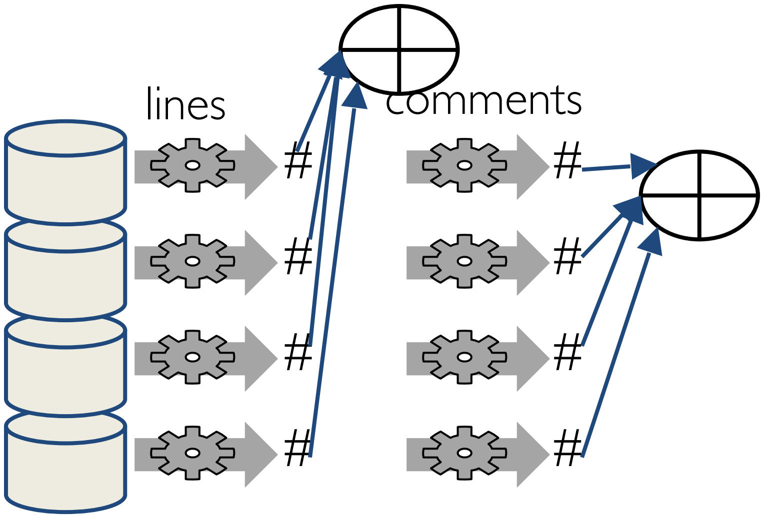


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark Programming Model

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

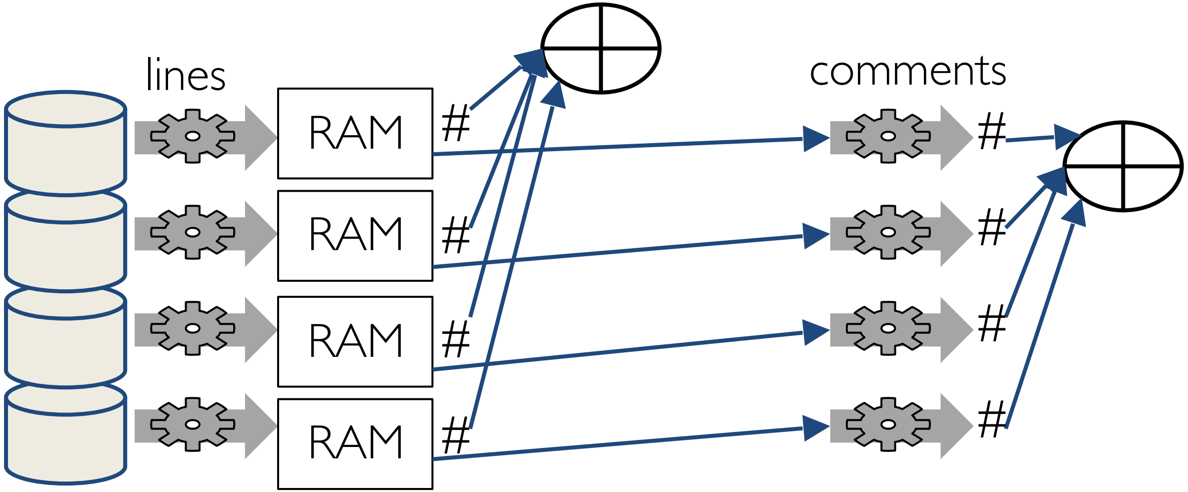


Spark recomputes **lines**:

- read data (again)
- sum within partitions
- combine sums in driver

Caching RDDs

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



Spark Program Lifecycle with RDDs

1. Create RDDs from external data or <u>parallelize</u> a collection in your driver program

2. Lazily <u>transform</u> them into new RDDs

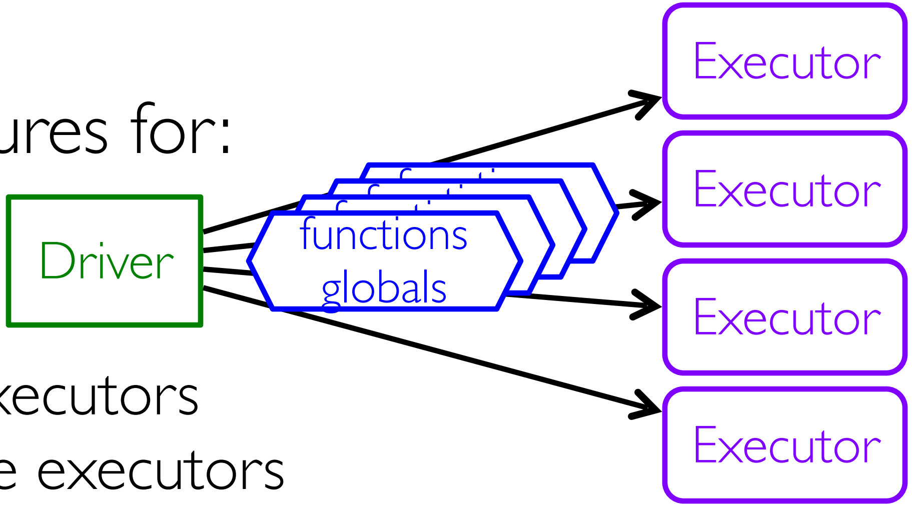
3. **`cache()`** some RDDs for reuse

4. Perform <u>actions</u> 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

pySpark Shared Variables

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

$\boxed{\Sigma}$

- » 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:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At an executor (in code passed via a closure)
>>> broadcastVar.value
[1, 2, 3]
```

[redacted]

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

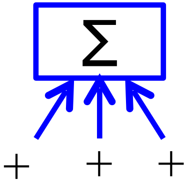
```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = sc.broadcast(loadCallSignTable())

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)))
```

Efficiently sent once to executors

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



Accumulators

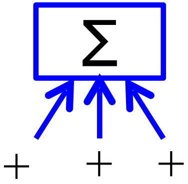
Variables that can only be "added" to by associative op

Used to efficiently implement parallel counters and sums

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

```
>>> 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
```



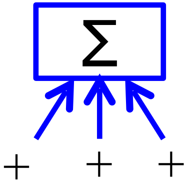
Accumulators Example

Counting empty lines

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

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

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



Accumulators

Tasks at executors cannot access accumulator's values

Tasks see accumulators as write-only variables

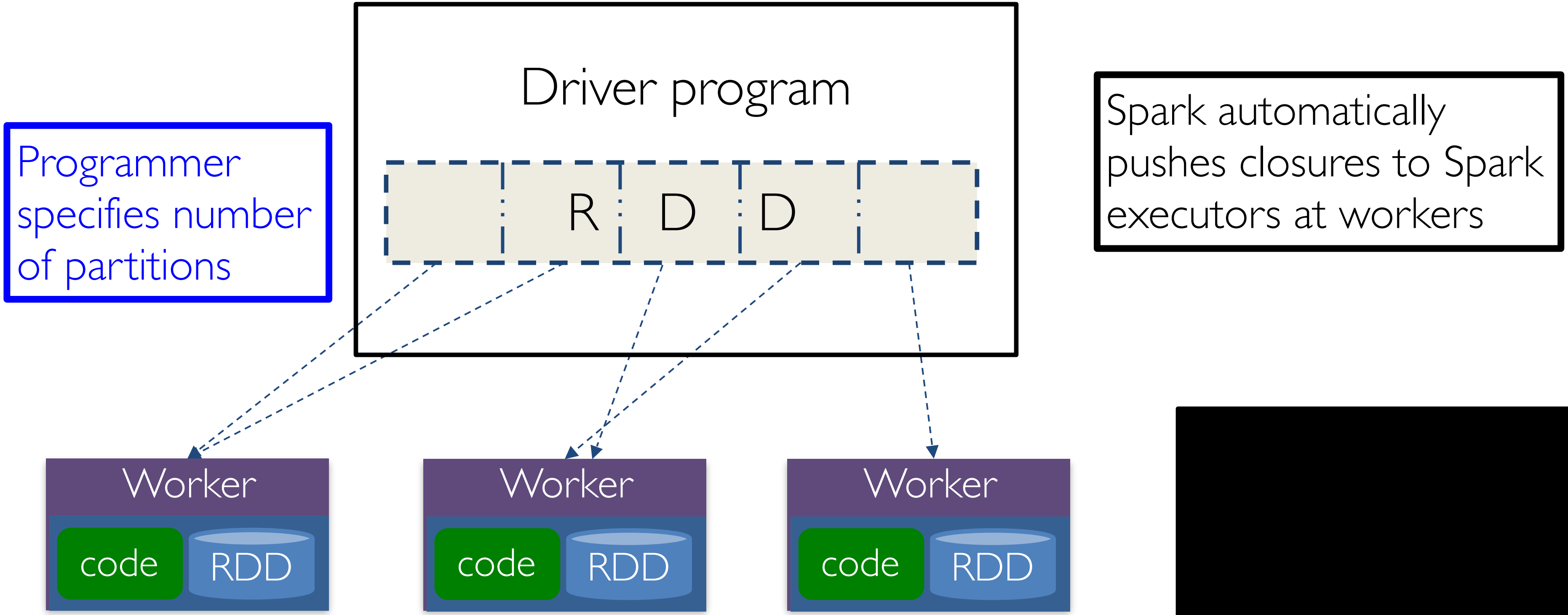
Accumulators can be used in actions or transformations:

- » Actions: each task's update to accumulator is ***applied only once***
- » Transformations: ***no guarantees*** (use only for debugging)

Types: integers, double, long, float

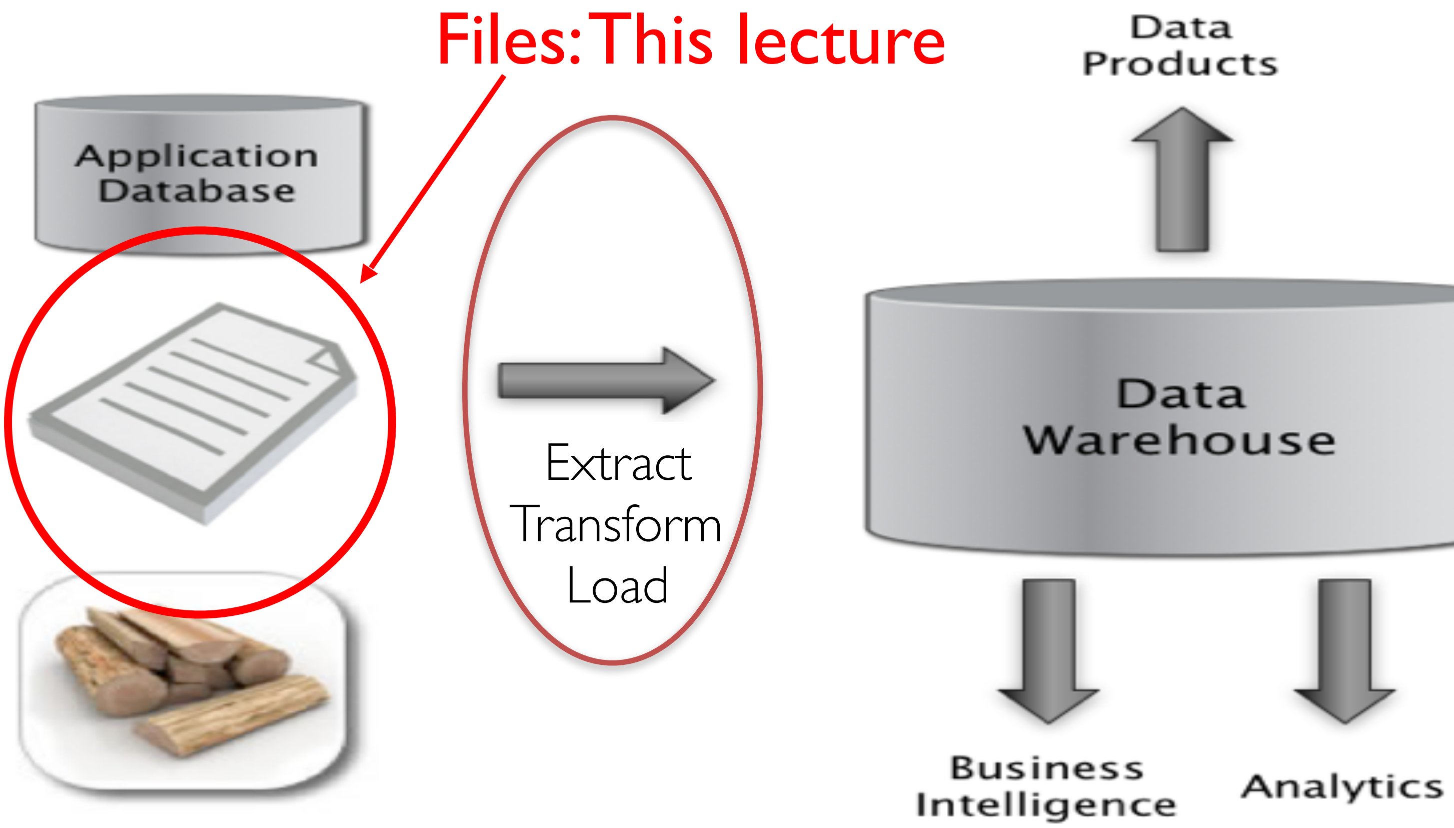
- » See lab for example of custom type

Summary



Master parameter specifies number of executors

Review: The Big Picture



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()`

http://www.clipshrine.com/Document-Icon-8906-medium.html

Files: Hierarchical Namespace

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

On Windows, \ is the root of a filesystem

Files and 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

626 MB text file
787 MB binary file

	Read Time (Text)	Write Time (Text)	Read Time (Binary)	Write Time (Binary)
Pandas (Python)	36 secs	45 secs	**	**
Scala/Java	18 secs	21 secs	1-6* secs	1-6* secs

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

626 MB text file
787 MB binary file

	Read Time (Text)	Write Time (Text)	Read Time (Binary)	Write Time (Binary)
Pandas (Python)	36 secs	45 secs	**	**
Scala/Java	18 secs	21 secs	1-6* secs	1-6* secs

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)

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 se	
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	423 MB
Raw binary file	1-6 secs	1-6 secs	787 MB

Write times much larger than read

Text File	Read Time	Write Time	File Size
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

[redacted]

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 secs	
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	423 MB
Raw binary file	1-6 secs	1-6 secs	787 MB

Large range of compression times

Text File	Read Time	Write Time	File Size
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

[redacted]

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 secs	
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	423 MB
Raw binary file	1-6 secs	1-6 secs	787 MB

Large range of compression times

Text File	Read Time	Write Time	File Size
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

[redacted]

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 secs	313 MB
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	423 MB
Raw binary file	1-6 secs	1-6 secs	787 MB

Small range (15%) of compressed file sizes

Text File	Read Time	Write Time	File Size
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 secs	313 MB
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	423 MB
Raw binary file	1-6 secs	1-6 secs	787 MB

Text File	Read Time	Write Time	File Size
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

Binary I/O still much faster than text

[redacted]

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 secs	313 MB
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	423 MB
Raw binary file	1-6 secs	1-6 secs	

Text File	Read Time	Write Time	
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

Binary I/O still much faster than text

[redacted]

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 secs	313 MB
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	[illegible]
Raw binary file	1-6 secs	1-6 secs	[illegible]

LZ4 compression ≈ raw I/O speed

Text File	Read Time	Write Time	File Size
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

[redacted]

File Performance - Compression

Scala/Java language

Binary File	Read Time	Write Time	File Size
Gzip level 6 (Java default)	4 secs	75 secs	286 MB
Gzip level 3	4 secs	20 secs	313 MB
Gzip level 1	4 secs	14 secs	328 MB
LZ4 fast	2 secs	4 secs	[illegible]
Raw binary file	1-6 secs	1-6 secs	[illegible]

LZ4 compression ≈ raw I/O speed

Text File	Read Time	Write Time	File Size
Gzip level 6 (default)	26 secs	98 secs	243 MB
Gzip level 3	25 secs	46 secs	259 MB
Gzip level 1	25 secs	33 secs	281 MB
LZ4 fast	22 secs	24 secs	423 MB
Raw text file	18 secs	21 secs	626 MB

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 ondvd-rom!product informationinspire your creativity and perfect any creative project with thousands ofworld-class images in virtually every style. plus clickart 950000 makes iteasy for …

Amazon listing:

- » clickart 950 000 - 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
  - = (# times *token* appears)/(total # of *tokens* in *doc*)

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*)

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:

Token	hello	goodbye	world
Count	1	1	2

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