Lecture 5
WordCount on XSEDE Jetstream

Instructor: Michela Taufer

THE UNIVERSITY OF
TENNESSEE
KNOXVILLE
BIG ORANGE. BIG IDEAS.®

Today Outline

- Zoom into WorkCount on Spark
  - What storage resources do we use and when?
  - How to run this and other benchmarks on the Cloud
- Projects
  - Two examples of projects: posters and 2-page abstract
- Assignments

WordCount Benchmark

In [4]

```
# Initializing Spark in Python
from pyspark import SparkContext, SparkConf
conf = SparkConf().setMaster('local').setAppName('WordCount')
sc = SparkContext(conf=conf)
```

WordCount Benchmark

In [4]

```
# Initializing Spark in Python
from pyspark import SparkContext, SparkConf
conf = SparkConf().setMaster('local').setAppName('WordCount')
sc = SparkContext(conf=conf)
```

cluster URL

application name

WordCount Benchmark

In [4]

```
# Initializing Spark in Python
from pyspark import SparkContext, SparkConf
conf = SparkConf().setMaster("local").setAppName("WordCount")
sc = SparkContext(conf=conf)


lines  = sc.textFile("FoxInSocks.txt")
words  = lines.flatMap(lambda line: line.split())
pairs  = words.map(lambda word: (word, 1))
counts = pairs.reduceByKey(lambda a, b: a+b) # counts is an RDD!
```

cluster URL

application name

WordCount Benchmark

In [4]

```
# Initializing Spark in Python
from pyspark import SparkContext, SparkConf
conf = SparkConf().setMaster("local").setAppName("WordCount")
sc = SparkContext(conf=conf)


lines  = sc.textFile("FoxInSocks.txt")
words  = lines.flatMap(lambda line: line.split())
pairs  = words.map(lambda word: (word, 1))
counts = pairs.reduceByKey(lambda a, b: a+b) # counts is an RDD!

results = counts.collect()
```

cluster URL

application name

WordCount Benchmark

In [4]

```
# Initializing Spark in Python
from pyspark import SparkContext, SparkConf
conf = SparkConf().setMaster("local").setAppName("WordCount")
sc = SparkContext(conf=conf)


lines  = sc.textFile("FoxInSocks.txt")
words  = lines.flatMap(lambda line: line.split())
pairs  = words.map(lambda word: (word, 1))
counts = pairs.reduceByKey(lambda a, b: a+b) # counts is an RDD!

results = counts.collect()
```

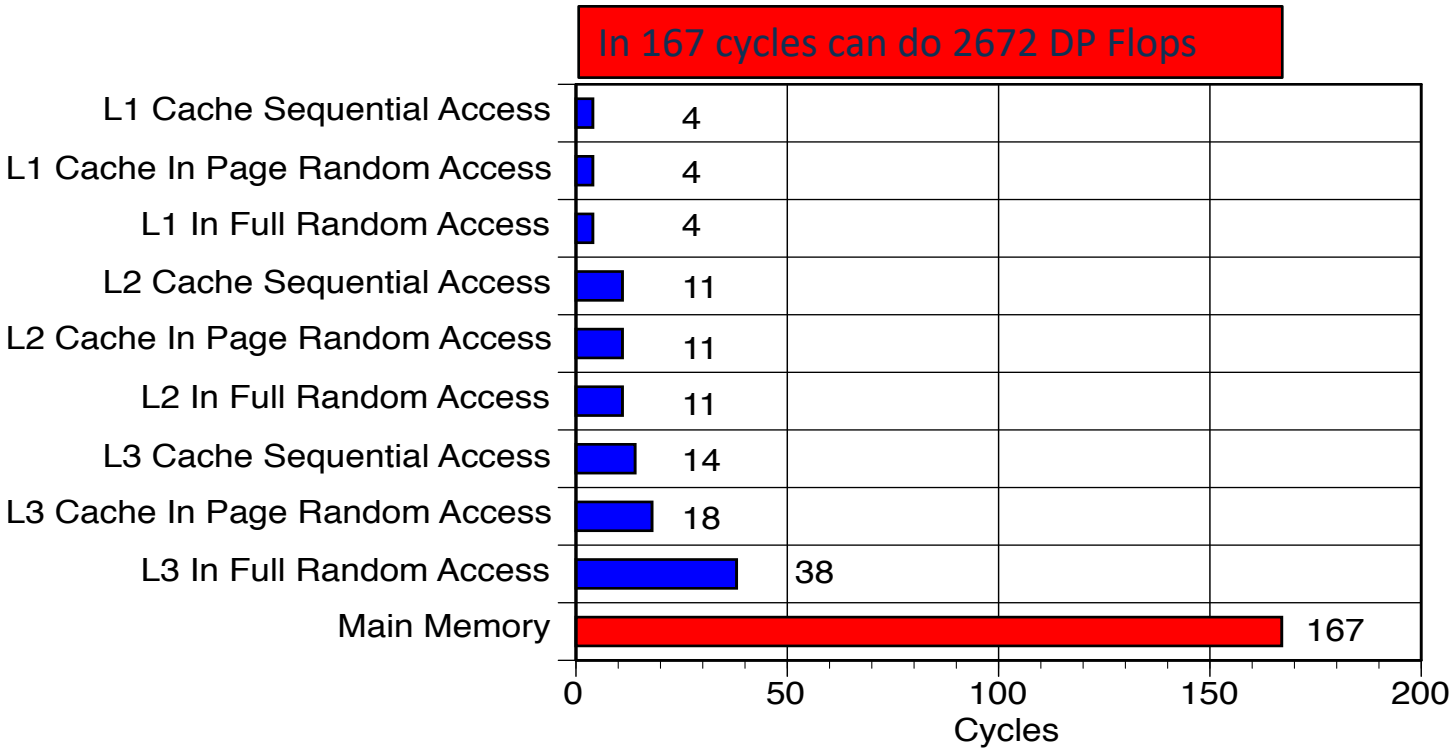
cluster URL

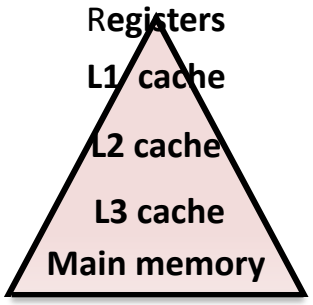
application name

Invoke the **action collect()** which brings all the elements of the RDD counts to the driver.
The action causes all the queued up transformations to be applied.

The Cost of Data Movement

- Today's floating point operations are inexpensive



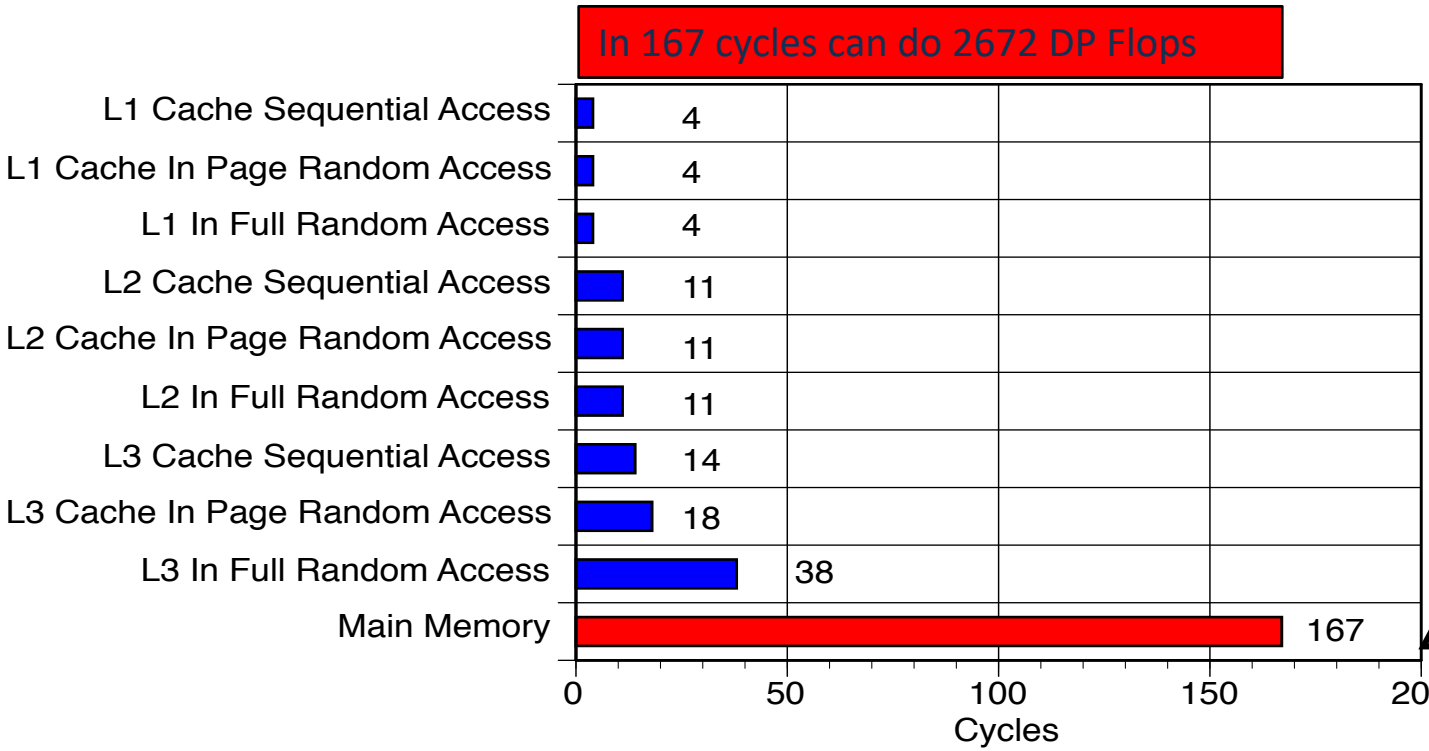


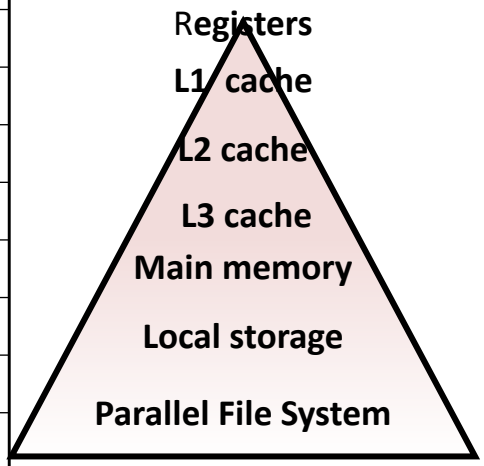
- Data movement is very expensive

Courtesy of Jack Dongarra, UTK and ORNL, 2017

The Cost of Data Movement

- Today's floating point operations are inexpensive



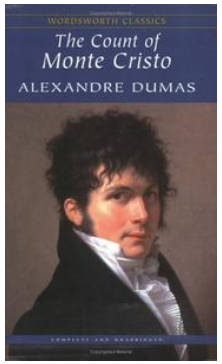


- Data movement is very expensive

Courtesy of Jack Dongarra, UTK and ORNL, 2017

What Is Happening in Memory?

```
lines = sc.textFile("Conte_of_Monte_Cristo.txt")
```

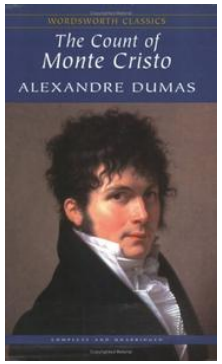


RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

No computation done.
Contents on disk not read into memory (yet).

What Is Happening in Memory?

```
lines = sc.textFile("Conte_of_Monte_Cristo.txt")
```

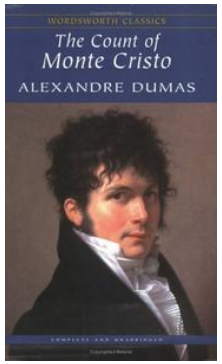


Spark doesn't create a new RDD **lines**.
Spark remembers how to create **lines**.
"If I am asked about **lines**, I will create it with **textFile(fileName)**."

RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

What Is Happening in Memory?

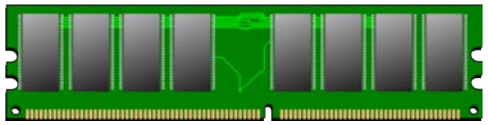
```
words = lines.flatMap(lambda line: line.split())
```



No computation done.
Contents on disk not read into memory (yet).

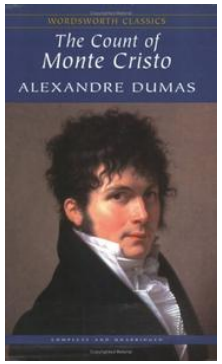
RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

RDD: words
size: 0 MB
location:
in memory



What Is Happening in Memory?

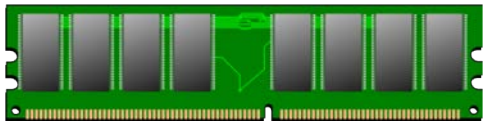
```
words = lines.flatMap(lambda line: line.split())
```



Spark doesn't create a new RDD **words**.
Spark remembers how to create **words**.
"If I am asked about **words**, I will apply **flatMap(func)** to **lines**."

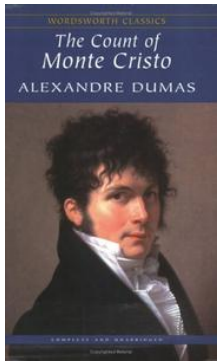
RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

RDD: words
size: 0 MB
location:
in memory



What Is Happening in Memory?

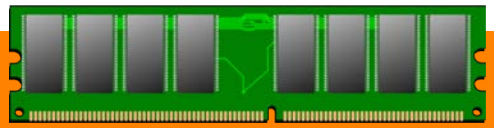
```
pairs = words.map(lambda word: (word, 1) )
```



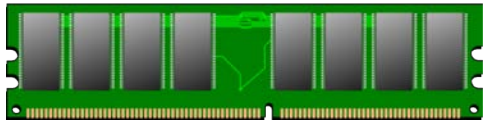
No computation done.
Contents on disk not read into memory (yet).

RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

RDD: words
size: 0 MB
location:
in memory

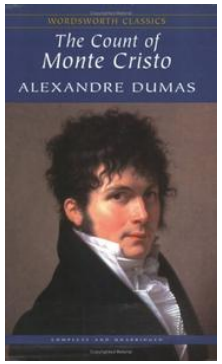


RDD: pairs
size: 0 MB
location:
in memory



What Is Happening in Memory?

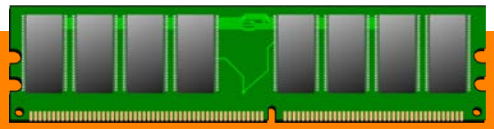
```
pairs = words.map(lambda word: (word, 1) )
```



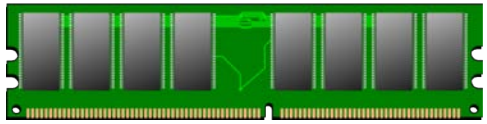
Spark doesn't create a new RDD **pairs**.
Spark remembers how to create **pairs**.
"If I am asked about **pairs**, I will apply **map(func)** to **words**."

RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

RDD: words
size: 0 MB
location:
in memory

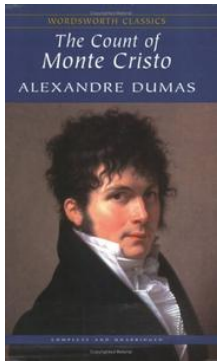


RDD: pairs
size: 0 MB
location:
in memory



What Is Happening in Memory?

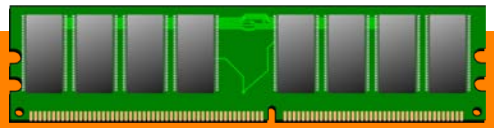
```
counts = pairs.reduceByKey(lambda a, b: a+b)
```



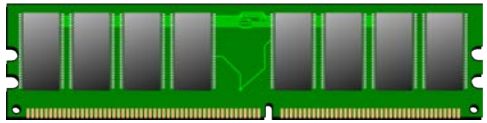
Still! No computation done.
Contents on disk not read into memory (yet).

RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

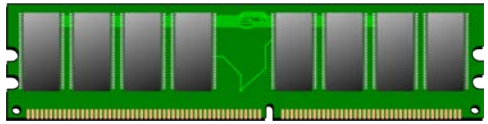
RDD: words
size: 0 MB
location:
in memory



RDD: pairs
size: 0 MB
location:
in memory

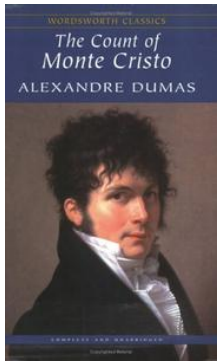


RDD: counts
size: 0 MB
location:
in memory



What Is Happening in Memory?

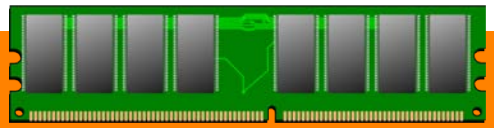
```
counts = pairs.reduceByKey(lambda a, b: a+b)
```



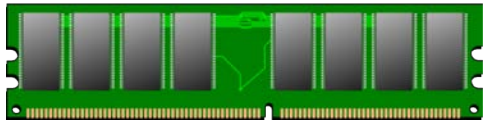
Spark doesn't create a new RDD **counts**.
Spark remembers how to create **counts**.
"If I am asked about **counts**, I will apply **reduceByKey()** to **pairs**."

RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

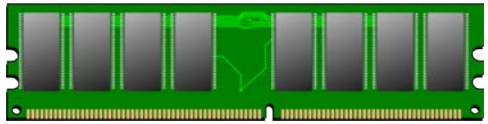
RDD: words
size: 0 MB
location:
in memory



RDD: pairs
size: 0 MB
location:
in memory

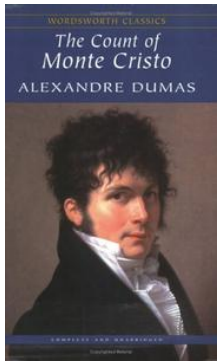


RDD: counts
size: 0 MB
location:
in memory



What Is Happening in Memory?

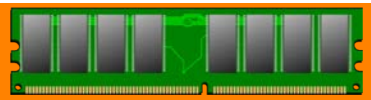
```
results = counts.collect()
```



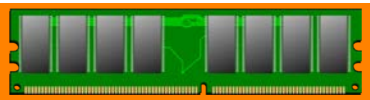
Now! Computation **must** be done.
The action **collect()** returns a list to the master process containing the elements of the RDD **counts**—Spark cannot return the values unless it first computes the values.

RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

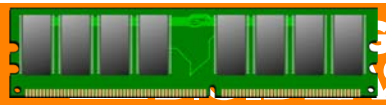
RDD: words
size: 0 MB
location:
in memory



RDD: pairs
size: 0 MB
location:
in memory

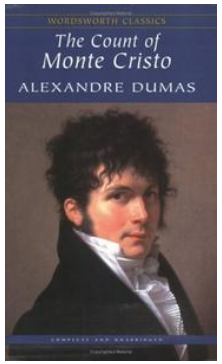


RDD: counts
size: 0 MB
location:
in memory



What Is Happening in Memory?

```
results = counts.collect()
```

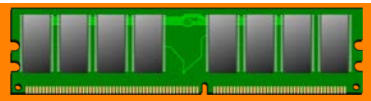


Now! Computation **must** be done.
The action **collect()** returns a list to the master process containing the elements of the RDD **counts**—Spark cannot return the values unless it first computes the values.

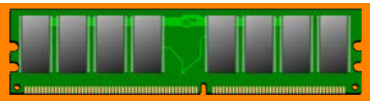
RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

flatMap()

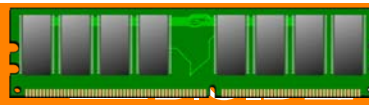
RDD: words
size: ~2.6 MB
location:
in memory



RDD: pairs
size: 0 MB
location:
in memory

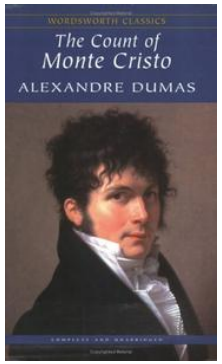


RDD: counts
size: 0 MB
location:
in memory



What Is Happening in Memory?

```
results = counts.collect()
```



Now! Computation **must** be done.
The action **collect()** returns a list to the master process containing the elements of the RDD **counts**—Spark cannot return the values unless it first computes the values.

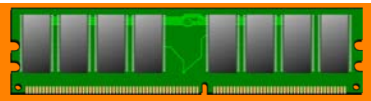
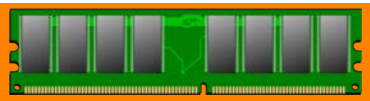
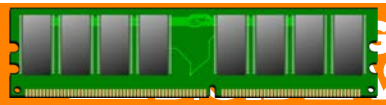
RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

RDD: words
size: ~2.6 MB
location:
in memory

map()

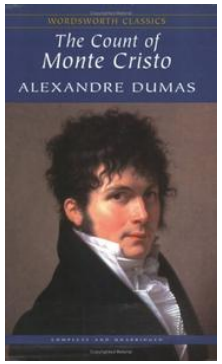
RDD: pairs
size: ~6 MB
location:
in memory

RDD: counts
size: 0 MB
location:
in memory

What Is Happening in Memory?

```
results = counts.collect()
```



Now! Computation **must** be done.
The action **collect()** returns a list to the master process containing the elements of the RDD **counts**—Spark cannot return the values unless it first computes the values.

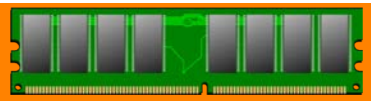
RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

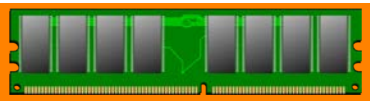
RDD: words
size: **?** MB
location:
in memory

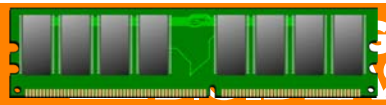
RDD: pairs
size: ~6 MB
location:
in memory

reduceByKey()

RDD: counts
size: ~1 MB
location:
in memory



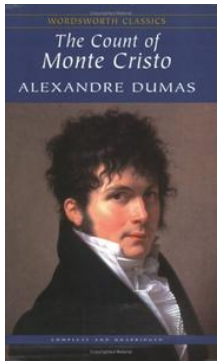




What Is Happening in Memory?

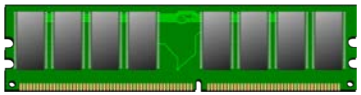
```
results = counts.collect()
```

Now! Computation **must** be done.
The action **collect()** returns a list to the master process containing the elements of the RDD **counts**—Spark cannot return the values unless it first computes the values.

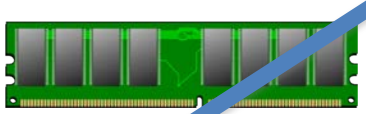


RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

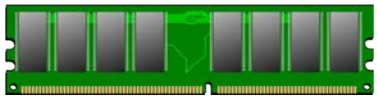
RDD: words
size: **?** MB
location:
in memory



RDD: pairs
size: **?** MB
location:
in memory

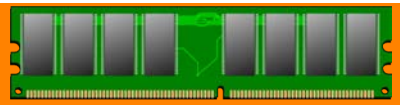


RDD: counts
size: **?** MB
location:
in memory

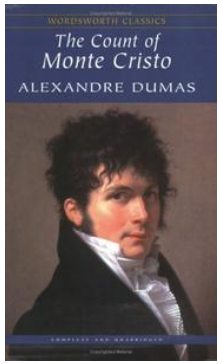


collect()

List: results
size: ~1 MB
Location:
in memory



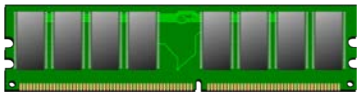
Little Details that Make All the Difference



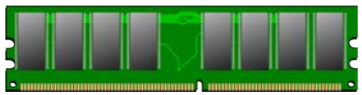
RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

Why ?

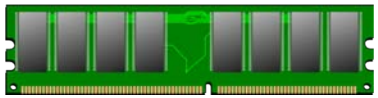
RDD: words
size: ? MB
location:
in memory



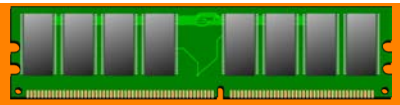
RDD: pairs
size: ? MB
location:
in memory



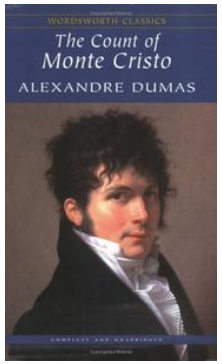
RDD: counts
size: ? MB
location:
in memory



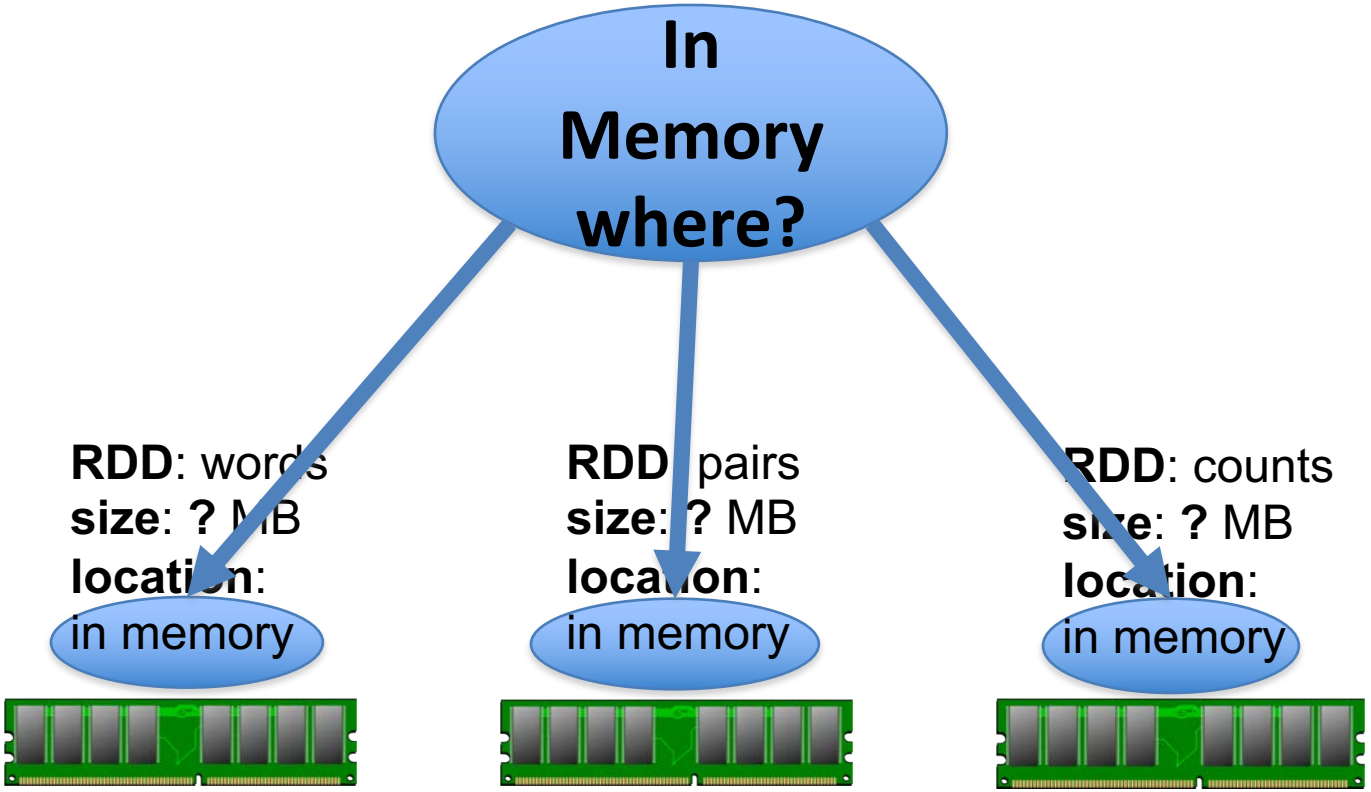
List: results
size: ~1 MB
Location:
in memory



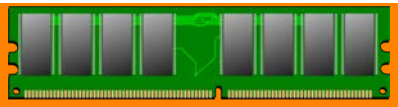
Little Details that Make All the Difference



RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

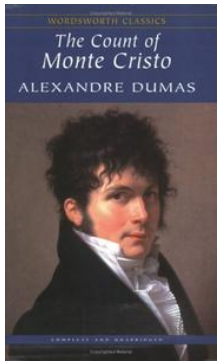


List: results
size: ~1 MB
Location:
in memory



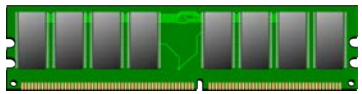
Little Details that Make All the Difference…

RDD's are **partitioned** and stored in **distributed memory**. If you have 100 nodes with 16 GB each, then an RDD could occupy up to 1,600 GB and still fit "in memory."

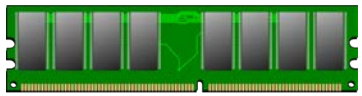


RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

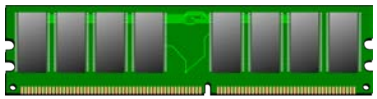
RDD: words
size: **?** MB
location:
in memory



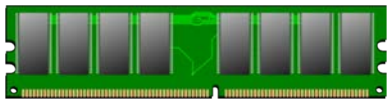
RDD: pairs
size: **?** MB
location:
in memory



RDD: counts
size: **?** MB
location:
in memory

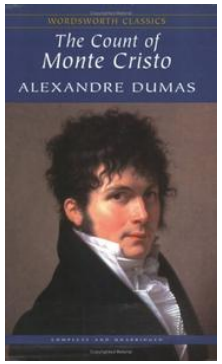


List: results
size: ~1 MB
Location:
in memory



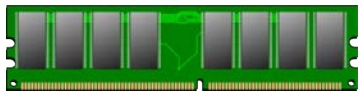
Little Details that Make All the Difference…

Results of **actions** are stored on the driver machine. These results are **not distributed**. If you have 100 nodes with 16 GB each, then the result of an action fits in memory **only if** it is smaller than 16 GB.

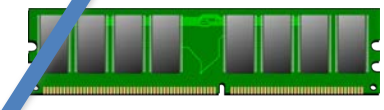


RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

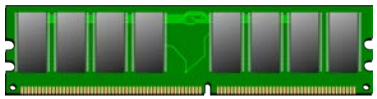
RDD: words
size: **?** MB
location:
in memory



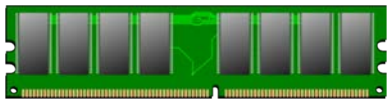
RDD: pairs
size: **?** MB
location:
in memory



RDD: counts
size: **?** MB
location:
in memory

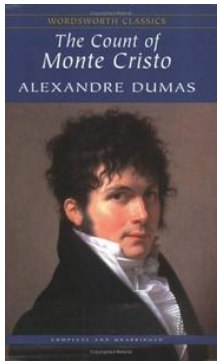


List: results
size: ~1 MB
Location:
in memory



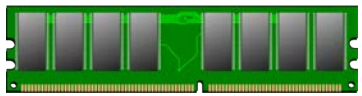
Little Details that Make All the Difference…

What happens to the intermediate RDDs after they were computed?

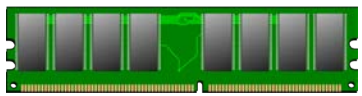


RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

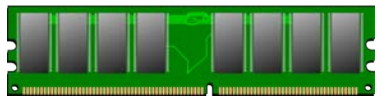
RDD: words
size: ? MB
location:
in memory



RDD: pairs
size: ? MB
location:
in memory



RDD: counts
size: ? MB
location:
in memory



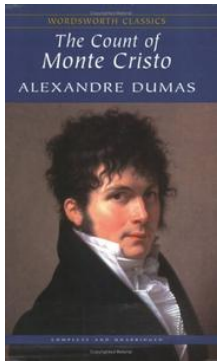
Little Details that Make All the Difference…

What happens to the intermediate RDDs after they were computed?

Spark **may** automatically cache the RDDs in memory, or it **may** re-compute them as necessary. **Unless…**

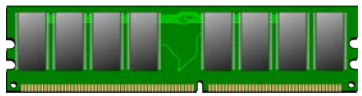
You insist that Spark cache the intermediate result:

- `counts.persist()`
- or, `counts.cache()`

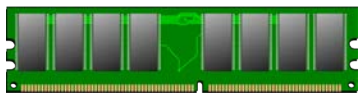


RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

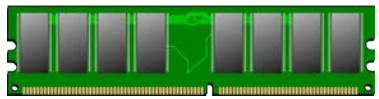
RDD: words
size: **?** MB
location:
in memory



RDD: pairs
size: **?** MB
location:
in memory



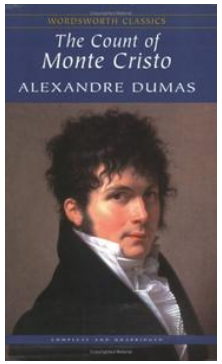
RDD: counts
size: **?** MB
location:
in memory



Little Details that Make All the Difference…

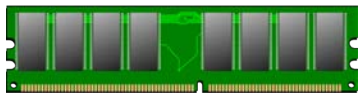
You insist that Spark cache the intermediate result:

- `counts.persist()` ← Allows you to specify the storage level for the RDD (MEMORY_ONLY, MEMEORY_AND_DISK, DISK_ONLY)
- or, `counts.cache()` ← Caches RDD at the default storage level (MEMORY_ONLY)

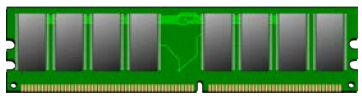


RDD: lines
size: 2.6 MB
location:
on disk (HDFS)

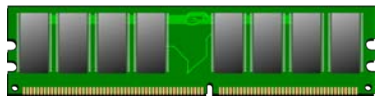
RDD: words
size: **?** MB
location:
in memory



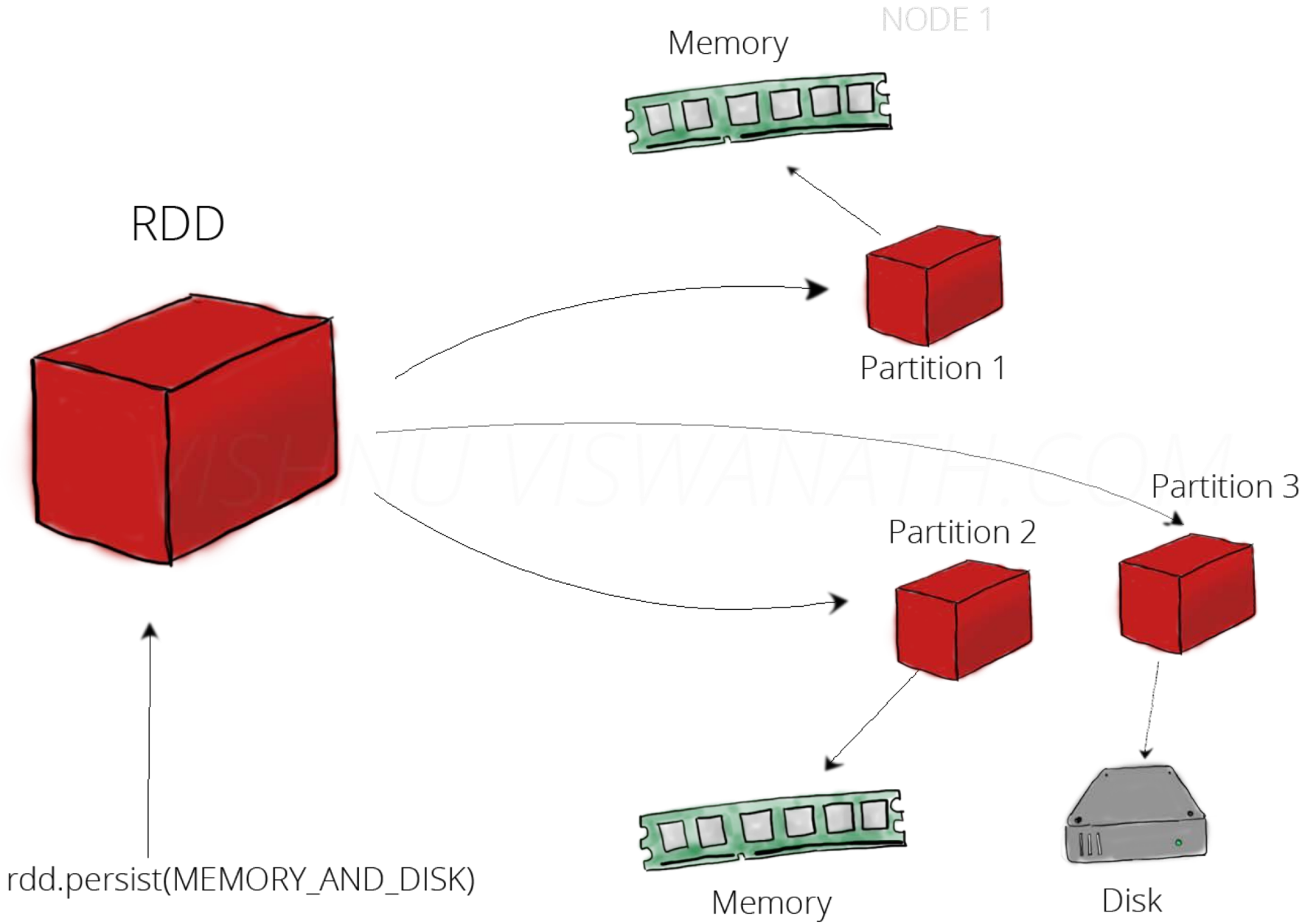
RDD: pairs
size: **?** MB
location:
in memory



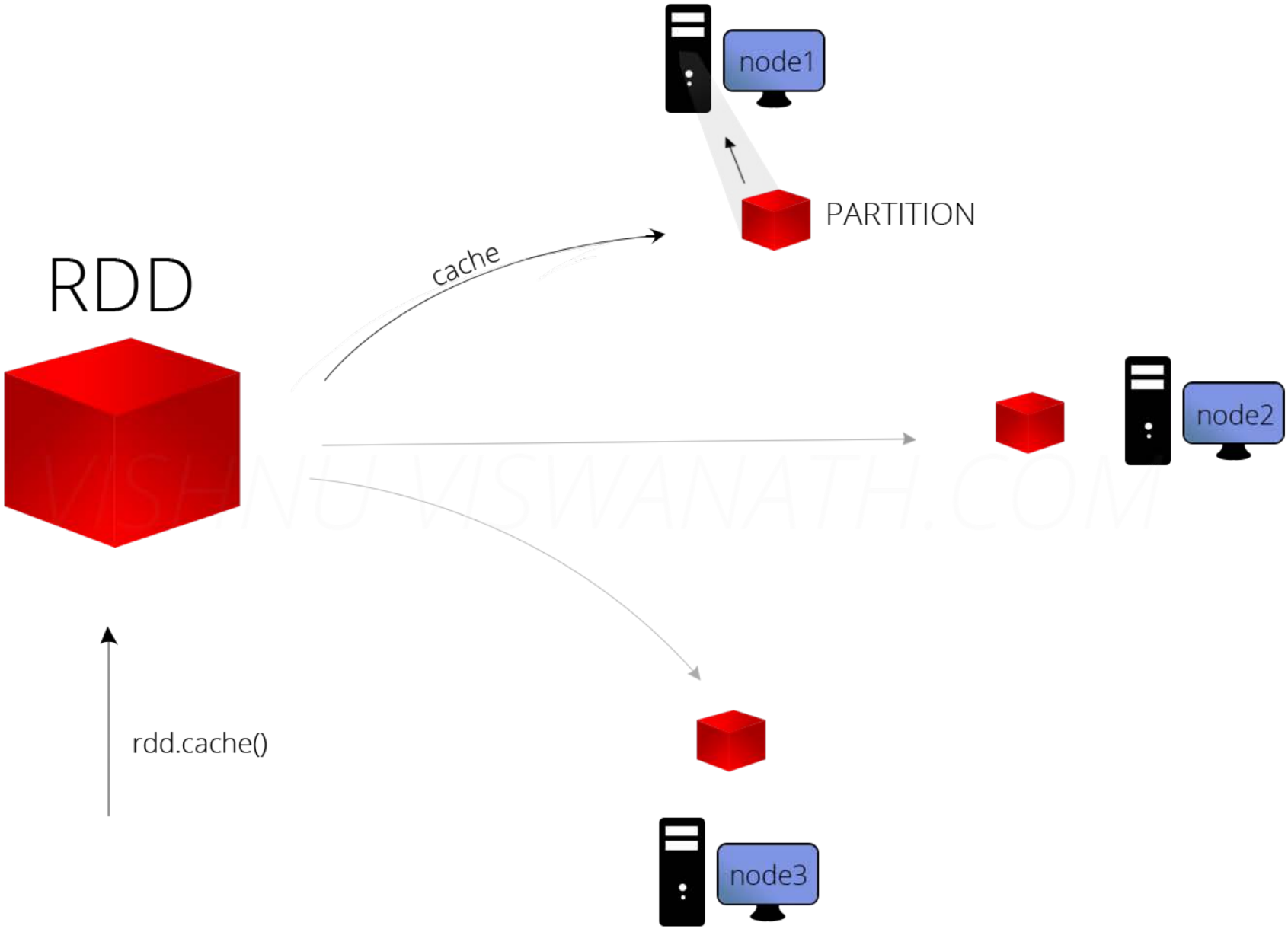
RDD: counts
size: **?** MB
location:
in memory



http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence



https://i.stack.imgur.com/cjD3K.png



https://i.stack.imgur.com/ekNcE.png

Rules of Thumb

- If your RDDs fit comfortably in memory (MEMORY_ONLY), leave them that way
  - This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible
- Do NOT spill to disk unless:
  - The functions that computed your datasets are expensive, or they filter a large amount of the data
  - Recomputing a partition may be as fast as reading it from disk

http://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

Project: Let's start ..

- Search for your dataset
- Examples:
  - Dylan's project was on a dataset from United States Federal Railroad Administration Office of Safety Analysis
  - Mike's project was on National Health and Nutrition Examination Survey (NHANES) dataset

Leveraging Spark and Docker for Scalable, Reproducible Analysis of Railroad Defects

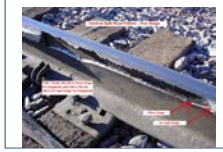
Dylan Chapp and Surya Kasturi
Advisors: Michela Taufer and Nii Attoh-Okine

Motivation

- Railroad network resiliency depends identification of defects
- Sensor-equipped monitoring cars collect rail and track-geometry data
- Can we use the data to predict defect occurrences in rail subdivisions?


Sensor-equipped Monitoring Car with its sensors

Rail and Track Defects Data Sets



- Rail defects data: physical degradation
  - 26,432 20-dimensional data points
- Track geometry data: misalignment
  - 25,421 41-dimensional data points
- Mixed numerical and categorical data

Scalable Reproducible Data Analysis Workflow

Track Defect Data

Rail Defect Data

Raw data provided by collaborators

Analysis Platform: PySpark + MLLib

Preprocessing
- Parallelized with Spark, scales to larger datasets
- Extracts logically related subsets of data
- Filters out records with missing data
- Filters out irrelevant feature columns

Targeted Subset of Data

Data is now ready to be ingested by an analysis tool, or directly visualized

MLLib Analysis Tools
- Descriptive Statistics
- Classification
- Clustering

Visualization
- Generate plots specific to analysis technique

Docker Container

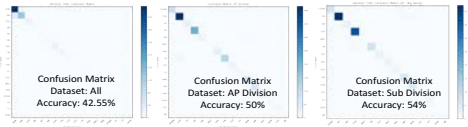
Input Data

Analysis Code

- Package data, scripts, configs as a Docker container
- Collaborators can easily reproduce our analysis
- Hosted on DockerHub: https://hub.docker.com/r/dchapp/rail

Predicting Defect Types

Can we predict defects in railroad tracks?
- Defects are classified using Decision Tree
- Defect size, accumulated tonnage, rail weight, rail section age are used as features

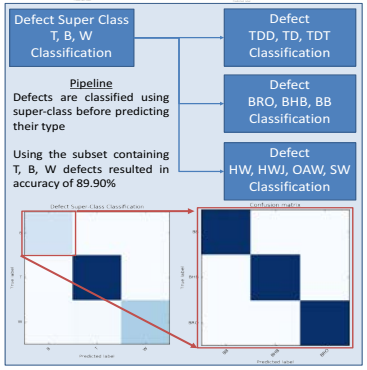


Defect Super Classes

T – Class
TDD, TDC, TD, TDT

B – Class
BRO, BHB, BB

W – Class
HW, HWJ, OAW, SW



Pipeline
Defects are classified using super-class before predicting their type

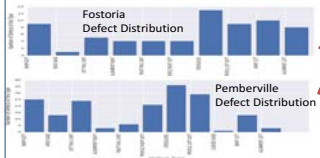
Using the subset containing T, B, W defects resulted in accuracy of 89.90%

Conclusions:
- We improve prediction accuracy by a hierarchical classification scheme
- First decide membership in defect superclass, then in defect class using a second classifier

Track Region Similarity Analysis

Can track regions be grouped so that defect-type classifiers trained on region-specific data achieve better accuracy?

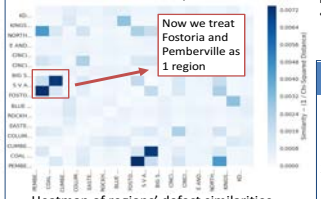
1: Extract each pair of regions' defect distributions



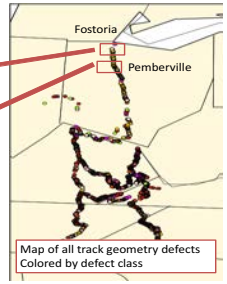
2: Compute the Chi-Squared Similarity of each pair of defect distributions

$$d(\mathbf{x}_u, \mathbf{x}_v) = \frac{1}{2}\sum_{n=1}^{N}\frac{[x_u(n) - x_v(n)]^2}{x_u(n) + x_v(n)}.$$

3: Group regions together whose defect distributions are sufficiently similar



Heatmap of regions' defect similarities



In progress:
- Training classifiers on subsets of defect data from statistically similar regions

References

1. A. Zarembski, "Some Examples of Big Data in Railroad Engineering", IEEE International Conference on Big Data, 2014
2. Track Inspector Rail Defect Reference Manual, Federal Railroad Administration, Rev. 2, 2015

Acknowledgements / funding

BIG ORANGE BIG IDEAS

Leveraging Spark and Docker for Scalable, Reproducible Analysis of Railroad Defects

Dylan Chapp
University of Delaware
dchapp@udel.edu

Surya Kasturi
University of Delaware
suryak@udel.edu

ABSTRACT

The resiliency of railroad networks depends on the ability of railroad engineers to identify and mitigate track and rail defects. As railroads modernize their defect identification measures, the volume and velocity of defect data substantially increases, necessitating adoption of techniques for "Big Data" analytics. We present a study of railroad defect prediction built atop Apache Spark and Docker to achieve scalability and reproducibility.

1. INTRODUCTION

According to the United States Federal Railroad Administration Office of Safety Analysis, track defects are the second leading cause of accidents on railways in the United States. In light of the economic significance of railway accidents [1], there is a pressing need in the railroad engineering community to adopt data-driven scalable data analysis tools from the greater "Big Data" ecosystem. [3] Track maintenance–i.e., identifying and repairing defects–is one of the primary factors that affect the service life of a rail track, but due to the severe safety implications of undetected or unprepaired defects, the ability to predict common defects is highly desirable.

In this work, we present a case study centered on the analysis of two railroad defect data sets obtained from railroad engineering researchers in the University of Delaware Department of Civil Engineering. Hereafter we will refer to these datasets as the `rail_defects` data set and the `track_geometry_defects` data set. Respectively, these data sets describe defects in the rails themselves, such as voids or internal changes in crystalline structure, and misalignment of track components, such as one rail tilting away from the other. [3] We investigate the feasibility of predicting the type of a defect based on associated data such as geographic region, mean gross tonnage (MGT) the track is subject to, and rail type. In the rest of this paper, we outline the construction of our analysis platform, present some initial results on classification accuracy, and propose extensions to our work.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. METHODOLOGY

Both of the data sets we target have mixed categorical and numerical features and $> 99\%$ of the individual defect records have a class label indicating the type of defect. In the case of `rail_defects`, there are 20 distinct defect types. For `track_geometry_defects`, there are 25 defect types. In light of these properties, we focus on the multilabel classification task for each data set. We decompose the task into a pipeline of three parts: preprocessing, training, and testing. We implement this pipeline using the MapReduce framework Apacahe Spark [2] and its parallel machine learning library MLLib, and package the data and analysis scripts as a Docker container for ease of dissemination. In the remainder of this section, we describe the pipeline components, also display in Figure 1

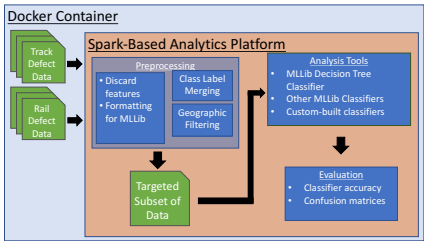


Figure 1: Block Diagram of Analytics Platform

We consider two stages of mandatory preprocessing. The first stage discards all columns except for a specified set, then discards any rows that are missing values for features from that set. The second stage maps the raw record strings to the format the MLLib API specifies, a key-value pair whose key is the type of defect and whose value is a feature vector.In addition to the above preprocessing, we implemented two optional stages: one to restrict the data to a geographically coherent region, and another to map each data point's class label to a "super- class" label indicating the general kind of defect (e.g., a welding-related defect, rather than one of the five kinds of specific welding defects). In our evaluation section, we demonstrate the usefulness of these additional preprocessing stages.

To build and evaluate our classifier, we split the subset of data remaining after preprocessing into training and testing sets consisting of, respectively, 70% and 30% of the original data. Membership in the training and testing sets is determined by uniform random sampling. We then train an instance of MLLib's decision tree classifier on the training set and test its predictions. In principle, any other MLLib classifier with a compatible API could be trained instead, but we elected to keep our classifier type fixed and investigate the effect of the "class-merging" and "geographic filtering" preprocessing steps on accuracy.

Table 1: Rail Defects Mapping

Super Class	Defect Types
T	TDD, TDC, TD, TDT
B	BRO, BHB, BB
W	HW, HWJ, OAW, SW
Others	SD, VSH, HSH, TW, CH, FH, PIPE, DR, EFBW

3. EVALUATION

To evaluate our classifier's performance, we examine the overall accuracy rate of the classifier and its associated confusion matrix. When we trained the classifier on training data drawn uniformly at random from `rail_defects` dataset with each defect type as a class label, the classifier predicted with an accuracy of 42.55%.

3.1 Class Label Merging

We propose mapping each data point's class label to a "super-class" label indicating its general kind. Out of 20 defect types in `rail_defects` dataset, 11 are mapped to 3 three "super-classes". Table 3.1 shows mapped and unmapped defect types.

With this mapping, we show that the prediction accuracy of rail defects is improved using a hierarchical classification scheme . First a classifier is trained to decide super-class of data point, then a second classifier is used to predict its defect type. When this model applied on the training data, the classifier predicted with an accuracy of 89.90%. Figure 2 shows the confusion matrix of the respective result.

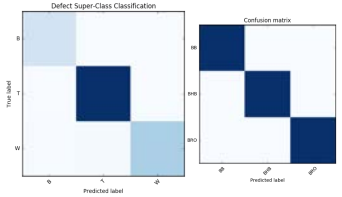


Figure 2: Confusion Matrix of the hierarchical classification scheme

3.2 Geographic Filtering

We propose that if subdivisions have similar numbers of each kind of defect, then we should group these subdivisions' data points and train a classifier with the expressed purpose of achieving good accuracy for that set of subdivisions. To determine which subdivisions to merge, we propose computing the χ-squared distance S defined below for each pair of subdivisions, then merge them based on a fixed threshold.

$$S(D_1, D_2) = \sum_{i=0}^{N} \frac{(x_i - y_i)^2}{(x_i + y_i}$$

We demonstrate the potential of grouping based on defect type distributions below. We compute $S(x, y)$ for each pair of subdivisions within the Appalachian division and display the results in the heat map in Figure 3.2

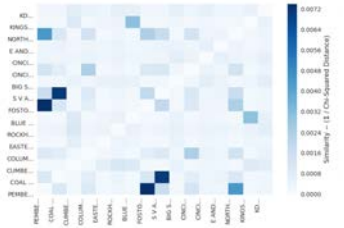


Figure 3: χ-squared distance between defect distributions for each subdivision

4. CONCLUSIONS AND CONTINUING WORK

We identified the potential of a hierarchical classification stage to improve the accuracy of defect type predictions. Additionally, we determined while that merely training classifiers on data from geographically-similar regions does not yield a significant improvement in accuracy, attempting to group together regions whose defect distributions are similar may prove useful.

Future directions for this work include evaluating classifiers beyond decision trees, and refining the similarity metric on defect distributions we use to group regions.

5. REFERENCES

[1] D. H. Schafer. A prediction model for broken rails and an analysis of their economic impact. *2008 AREMA Conference*, 2008.

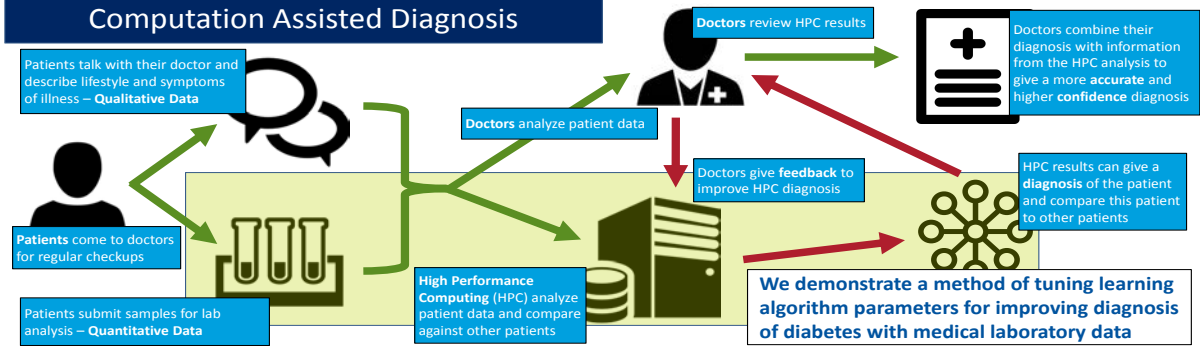
[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

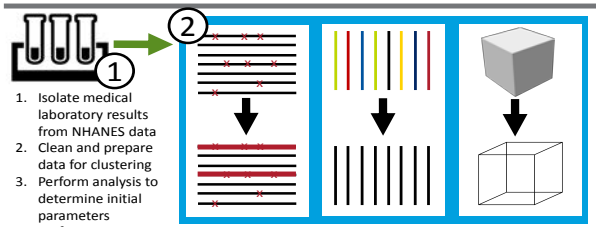
[3] A. M. Zarembski. Some examples of big data in railroad engineering. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 96–102, Oct 2014.

Parameter Tuning of DBSCAN for Medical Data and Diabetes Diagnosis

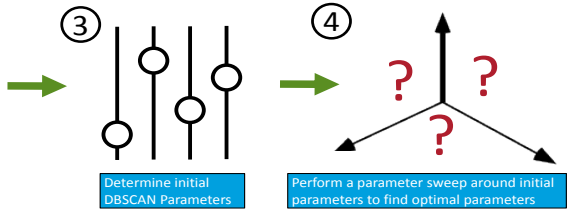
Michael Wyatt, Michela Taufer
University of Delaware: Global Computing Lab

Computation Assisted Diagnosis





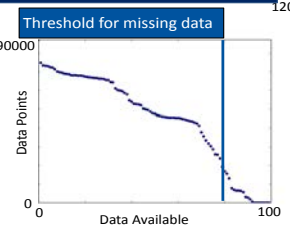
Parameter Tuning Workflow



Parameter Tuning Results

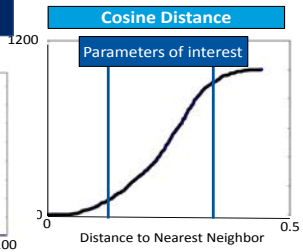
- Missing data **trade-off**
  - No Missing Values → **Small Dataset**
  - Many Missing Values → **Bad Clusters**

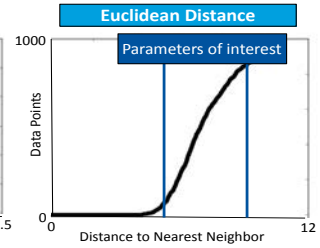
  Must find a balance between missing values and dataset size
- Patients with > 80% lab data available
  - 16,627 patients (over 1/5 original data)
  - Produces higher quality clusters



- DBSCAN parameters affect cluster quality
  - **Epsilon**: Neighborhood to search for neighbors
  - **Min_pts**: minimum neighbors to be in a cluster

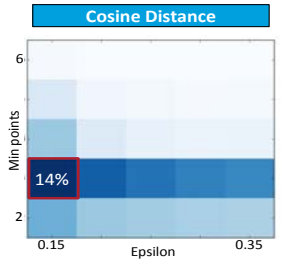
  } Parameters define cluster **density**
- Distance metrics also affect cluster quality: **Euclidean** vs. **Cosine**

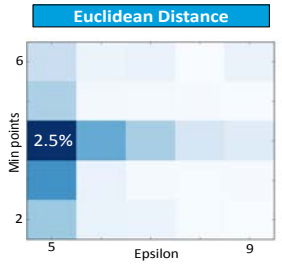




- Utilizing nearest neighbor analysis, we can determine the range of epsilon values which should be tested
- We cluster data with several **epsilon** and **min_pts** values around the identified optimal values
- We measure the quality of each clustering by **percentage of points clustered** and **information gained** by each clustering:

$$score = \frac{Diebetes\ Patients\ Clustered}{Total\ Diabetes\ Patients} * \frac{Informaion\ Gain}{Max\ Information\ gain}$$





- We identify an optimal parameter setting:
  - **Cosine distance, Epsilon: 0.15, Min_pts: 3**

BIG ORANGE BIG IDEAS

Parameter Tuning of DBSCAN for Medical Data and Diabetes Diagnosis

[Extended Abstract]

Michael R. Wyatt II
University of Delaware
18 Amstel Ave
Newark, DE 19701
mwyatt@udel.edu

Michela Taufer
University of Delaware
18 Amstel Ave
Newark, DE 19701
taufer@udel.edu

ABSTRACT

The increasing use of computationally assisted diagnosis in the doctor's office requires that computer diagnosis be both fast and accurate. We present a scalable method for preparing laboratory data for use with learning algorithms and a method for identifying optimal parameter settings for learning algorithms. To demonstrate our method, we predict the presence of diabetes among participants of the National Health and Nutrition Examination Survey using collected laboratory data and the DBSCAN algorithm. We performed optimization of the DBSCAN parameters for this dataset to demonstrate how diagnosis predictions can be improved.

CCS Concepts

•Applied computing → Health care information systems; *Consumer health;* **•Computing methodologies →** *MapReduce algorithms;*

Keywords

Health Informatics; Machine Learning; Optimization

1. MOTIVATION

Modern medical diagnosis is becoming increasingly computationally assisted. This means that artificial intelligence and machine learning algorithms are being used to analyze patient medical data and provide a diagnosis. Human doctors consult the computation results and a final diagnosis is made. As computationally assisted diagnosis becomes more widespread, patient diagnosis becomes more accurate [1] [3]. An important aspect of computationally assisted diagnosis is the processing of medical data by learning algorithms to produce useful results. Improving the speed and accuracy of this process will encourage the continued adoption of computationally assisted diagnosis by medical doctors, which will lead to improved population health and disease management.

2. CONTRIBUTIONS

Many efforts have been made to improve both the accuracy and processing time of computationally assisted diagnosis. These efforts focus mainly on the application of different algorithms to medical data. In this paper, we apply the clustering algorithm DBSCAN to medical data in order to diagnose patients with diabetes. We present a framework for parallel processing of medical data for learning algorithms. Additionally, we outline a method for optimizing learning algorithm parameters to achieve optimal results.

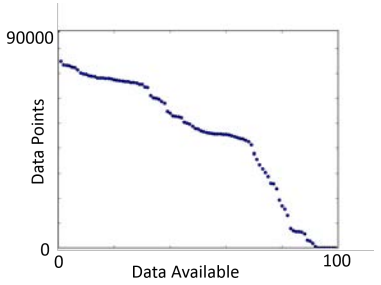


Figure 1: NHANES participants sorted by percent of laboratory data available.

3. METHODOLOGY

We isolate lab data from the National Health and Nutrition Examination Survey (NHANES) dataset for over 70,000 participants. We process this data using parallel algorithms built with the MapReduce programming paradigm via Apache Spark. The processing of data is highly scalable across many nodes. The processed data is in a form that is usable by learning algorithms like DBSCAN. We then perform parameter optimization to achieve maximum predictive capabilities.

3.1 NHANES Dataset

We obtained data from the NHANES continuous dataset collected between 1999 and 2014. We label the participants as having or not having diabetes based on their categorical response to the question, "have you ever been told by a doctor or health professional that you have diabetes or sugar diabetes?" We isolate 116 common features within the laboratory data, including urine and blood sample values, which can be used to cluster the diabetic and non-diabetic NHANES participants.

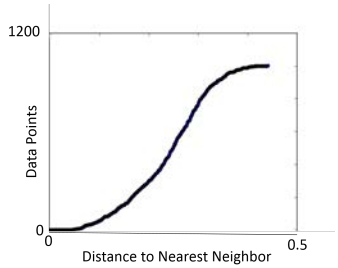


Figure 2: Number of data points (y axis) with a neighbor within distance *Epsilon* (x axis). The elbow in the plot indicates optimal values for *Epsilon*.

3.2 Data Preparation

We prepare the data by removing participants with many missing values, standardizing the data by feature, and dimensionality reduction. Figure 1 plots the participants sorted by the percent of data points (from the 116 features) available. All NHANES participants had missing values and many had most laboratory values missing. We identify a subset of 16,627 participants with more than 80% of features available for analysis. Each of the 116 features are standardized using Z-score standardization. This process makes the range of values for each feature similar to prevent one feature outweighing others (due to a large range of values). Dimensionality reduction is performed with Principal Component Analysis (PCA). The processing of NHANES data is performed with MapReduce algorithms. This allows the process to be distributed across many compute nodes and reduces result turn-around time.

3.3 Choosing Initial Parameters

There are three parameters for DBSCAN. Together, they define the density of clusters which will be found.

1. *Distance* - Metric used for calculating distance between patients
2. *Epsilon* - Distance around patients to identify neighboring patients
3. *Min_pts* - Number of neighboring patients to be "core" point

Like other learning algorithms, these parameters affect the quality of results. We chose to test two distance metrics: Euclidean and Cosine. We define cosine distance in equation 1. A range of *Min_pts* values was selected for testing. We then determined values for *Epsilon* by adapting the "elbow method" used for determining the best value of k in k-Means clustering [2]. Figure 2 shows the sorted Cosine distance to the nearest neighbor for each participant. We propose that ideal values for *Epsilon* will be around the elbow of this figure. In the case of figure 2, this range is $[0.15, 0.35]$.

$$Cosine\_distance(x, y) = 1 - \frac{x \cdot y}{\|x\|\|y\|} \quad (1)$$

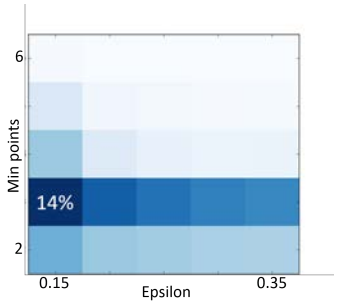


Figure 3: Information gain scoring method for Cosine distance metric across several *Epsilon* and *Min_pts* values.

3.4 Evaluation

To evaluate DBSCAN performance, we developed a scoring metric based on information gain. This scoring metric, seen in equation 2, considers the amount of information gain and size of each cluster in order to produce a value between 0 and 1. We scored each set of DBSCAN parameters and compared scores to find the best settings.

$$score = \frac{ClusteredPatients}{TotalPatients} \cdot \frac{InformationGain}{MaxInformationGain} \quad (2)$$

4. RESULTS

We observed that Cosine distance scored much higher than Euclidean distance. Figure 3 shows the scores of our parameter sweep around initial parameter selections for the Cosine distance metric. There is a clear set of parameters at $Epsilon = 0.15$ and $Min\_pts = 3$ which provides the best clustering score. The maximum observed score from our testing was 14%. Observed scores from our clustering were unexpectedely low. Despite the poor ability of DBSCAN to differentiate between patients with and without diabetes, our method of focused parameter searching was able to find optimal parameter settings.

5. REFERENCES

[1] K. Doi. Computer-aided diagnosis in medical imaging: historical review, current status and future potential. *Computerized medical imaging and graphics*, 31(4):198–211, 2007.

[2] T. M. Kodinariya and P. R. Makwana. Review on determining number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.

[3] I. Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.

Assignment 5

- Assignment 5 is completely performed on Jetstream account
  - Open your Jetstream account
  - Clone your solution of Assignment4
  - Clone Assignment 5
- Helpful scripting:
  - Use preinstallenv.ipynb to set up your environment on Jetstream
  - Use preinstallenv.ipynb on your local machine

Assignment 5

- Problem 0: run your solution of Assignment 4 on Jetstream
- Problems 1- 3: use pyspark to solve the same problems you already solved sequentially in Assignment 4 – this time you run distributed solutions
- Use matplotlib to create histograms for Problems 1-3
- Problem 5: search for datasets and reflect on what you have learned today

DUE DATE: October 8, 2018 – 8AM ET

THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

BIG ORANGE. BIG IDEAS.®

BIG ORANGE BIG IDEAS