CSE 4/587
Data Intensive Computing

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Dr. Shamshad Parvin
shamsadp@buffalo.edu
313 Davis Hall

Introduction to Spark
Announcements

● Midterm grading is wrapping up
● HW #2 is coming soon...
  ○ All about Spark, if you want to get a headstart, install pyspark
    https://spark.apache.org/docs/latest/api/python/getting_started/install.html
References

- **Advanced Analytics with Spark** by S. Ryza, U. Laserson, S. Owen and J. Wills
- **Apache Spark documentation**
- **Pyspark**
  - [https://spark.apache.org/docs/latest/api/python/getting_started/install.html](https://spark.apache.org/docs/latest/api/python/getting_started/install.html)
- **Resilient Distributed Dataset: A Fault-tolerant Abstraction for in-Memory Cluster Computing.** M. Zaharia et al.
Challenges

Data cleaning: Majority of the work that goes into analyses lies in pre-processing data

- Munging, fusing, mushing and cleansing
- We need computational methods to clean data and data pipeline certainly should include an important step of “data cleaning” and “feature engineering”.

- Choosing from many features, the relevant features.
- Designing a math model from a 2D array (Ex: page rank)
Challenges

**Data cleaning:** Majority of the work that goes into analyses lies in pre-processing data
- Munging, fusing, mushing and cleansing
- We need computational methods to clean data and data pipeline certainly should include an important step of “data cleaning” and “feature engineering”.
- Choosing from many features, the relevant features.
- Designing a math model from a 2D array (Ex: page rank)

*Need to avoid delays in repeated reading of data*
Challenge: Iteration is a fundamental part of data science.

- Modeling and analysis require typically multiple passes over the same data.
- Machine learning algorithms and statistical procedures like stochastic gradient and expected maximization involve repeated scans to reach convergence.
- Choosing the right features, picking the right algorithms, running the right significance tests, finding the right hyperparameters: all require experimentation.
Challenges

**Iteration:** Iteration is a fundamental part of data science.
- Modeling and analysis require typically multiple passes over the same data
- Machine learning algorithms and statistical procedures like stochastic gradient and expected maximization involve repeated scans to reach convergence
- Choosing the right features, picking the right algorithms, running the right significance tests, finding the right hyperparameters: all require experimentation

Need to avoid delays in repeated reading of data
Information updates: The results of data analysis presented and the application becomes part of the production system...

- This system must frequently or in real time update itself driven by the availability of new data; i.e., fraud detection system.
Challenges

Information updates: The results of data analysis presented and the application becomes part of the production system...

- This system must frequently or in real time update itself driven by the availability of new data; ie fraud detection system.

How about the existing approaches?

- C++, Java are not good for EDA
- R is slow for large data sets and does not integrate well with production stacks
- Read-Evaluate-Print-Loop (REPL) are good for interaction but not work production
Challenges

Information updates: The results of data analysis presented and the application becomes part of the production system...
- This system must frequently or in real time update itself driven by the availability of new data; ie fraud detection system.

How about the existing approaches?
- C++, Java are not good for EDA
- R is slow for large data sets and does not integrate well with production stacks
- Read-Evaluate-Print-Loop (REPL) are good for interaction but not work production

Want a framework that makes modeling easy, but also fits well in production systems
Apache Spark is an open-source, distributed processing system commonly used for big data workloads.  
- Utilizes in-memory caching  
- Optimized execution for fast performance  
- Supports general batch processing, streaming analytics, machine learning, graph databases, and ad hoc queries
Spark vs MapReduce

MapReduce offers linear scalability and fault tolerance for processing very large data sets.

Spark maintains this revolutionary approach brought about by MapReduce.

It also improves it in four different ways:

- Executes a series of operations specified in a DAG
  - Allows one stage of "MR" to send the results to the next (Similar to Microsoft Dryad)
- Provides a rich set of operations to express computation more naturally (Similar to Pig)
- Improves on in-memory computations through its Resilient Distributed Dataset (RDD)
  - Future steps dealing with the same data do not have reload it from the disk
- Well-suited for highly iterative computing
Sample of Performance Results

![Bar chart showing iteration times for different systems and algorithms.](chart.png)

- **Logistic Regression**:
  - Hadoop: 80s
  - HadoopBM: 76s
  - Spark: 62s
  - First Iteration: 46s (Spark)

- **K-Means**:
  - Hadoop: 139s
  - HadoopBM: 139s
  - Spark: 87s
  - First Iteration: 182s
  - Later Iterations: 106s (HadoopBM)
Biggest bottleneck in data applications is not CPU, disk, or network but analyst productivity
Biggest bottleneck in data applications is not CPU, disk, or network but analyst productivity.

If only we could collapse the entire pipeline from pre-processing of data to model evaluation into a single programming environment...

Spark transitions seamlessly between exploratory analytics and operational analytics.
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

It also provides APIs in many common languages:

- Scala API
- Java API
- Python API
- Dataframes API
- R API
Spark's Python API allows the use of lambda functions to transform data

```python
x = lambda a : a + 10
print(x(5))

x = lambda a, b : a * b
print(x(5, 6))
```
Spark Ecosystem

Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)

Apache Spark
Spark Architecture
Programming Model

- **Spark Context**: sc
- **RDD**: Resilient Distributed Datasets
  - Transformations and actions on RDDs
- **Diverse set of data sources**: HDFS, relational databases
- **Diverse APIs**: Java API, Python API, Scala API, Dataframe API, R API
Spark Context

Spark Context (sc) is an object
- Main entry point for Spark applications
- Just like any object it has methods associated with it
- Some of the methods:
  - getConf
  - runJob
  - addFile
  - cancelAllJobs
  - makeRDD
A core Spark concept is **Resilient Distributed Datasets (RDD)** which is a fault tolerant collection of elements that can be operated on in parallel.

An RDD is a convenient way to describe the computations that we want to perform in small independent steps and in parallel.

**There are two ways to create RDDs:**
- Parallelizing an existing collection in the driver program; performing a transformation on one or more existing RDDs, like filtering records, aggregating records by a common key or by joining multiple RDDs together.
- Using SparkContext to create an RDD from an external dataset in an external storage system such as a shared filesystem, HDFS, HBase or any data source offering a Hadoop input format.
Examples

A few transformations to build a dataset and store into a file:

text-file = sc.textFile("hdfs://...")

counts = text_file.flatMap(lambda line: line.split(" "))
  .map(lambda word: (word,1))
  .reduceByKey(lambda a,b: a+b)

counts.saveAsTextFile("hdfs://..")
Resilient Distributed Datasets (RDDs)

The building block of the Spark API
(http://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds)

In RDD API there are two types of operations:
1. *Transformations* that define a new data set based on previous ones
2. *Actions* which kick off a job to execute on a cluster
RDD Transformations and Actions

Transformations:
- map(func)
- flatMap(func)
- filter(func)
- groupByKey()
- reduceByKey(func)
- mapValues(func)
- sample(...)
- union(other)
- distinct()
- sortByKey()
- ...

Actions:
- reduce(func)
- collect()
- count()
- first()
- take(n)
- saveAsTextFile(path)
- countByKey()
- foreach(func)
- ...
<table>
<thead>
<tr>
<th>Transformations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(f : T ⇒ U)</td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td>filter(f : T ⇒ Bool)</td>
<td>RDD[T] ⇒ RDD[T]</td>
</tr>
<tr>
<td>flatMap(f : T ⇒ Seq[U])</td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td>sample(fraction : Float)</td>
<td>RDD[T] ⇒ RDD[T] (Deterministic sampling)</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</td>
</tr>
<tr>
<td>reduceByKey(f : (V, V) ⇒ V)</td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td>union()</td>
<td>(RDD[T], RDD[T]) ⇒ RDD[T]</td>
</tr>
<tr>
<td>join()</td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</td>
</tr>
<tr>
<td>cogroup()</td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W])])</td>
</tr>
<tr>
<td>crossProduct()</td>
<td>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</td>
</tr>
<tr>
<td>mapValues(f : V ⇒ W)</td>
<td>RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)</td>
</tr>
<tr>
<td>sort(c : Comparator[K])</td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td>partitionBy(p : Partitioner[K])</td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>RDD[T] ⇒ Long</td>
</tr>
<tr>
<td>collect()</td>
<td>RDD[T] ⇒ Seq[T]</td>
</tr>
<tr>
<td>reduce(f : (T, T) ⇒ T)</td>
<td>RDD[T] ⇒ T</td>
</tr>
<tr>
<td>lookup(k : K)</td>
<td>RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)</td>
</tr>
<tr>
<td>save(path : String)</td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.
Resilient Distributed Datasets (RDDs)

A **distributed memory abstraction** that enables **in-memory computations** on large clusters in a **fault-tolerant** manner

- **Motivation**: iterative algorithms, interactive data mining tools
  - In both cases above keeping data in memory will help enormously for performance improvement
- **RDDs** are parallel data structures allowing coarse grained transformations
- **It provides fault tolerance by storing the lineage as opposed to the actual data** as done in Hadoop
Transformations vs Actions

- **Transformations are lazy**: nothing actually happens when this code is evaluated.
- **RDDs are computed only when an action is called on them**, e.g.:
  - Calculate statistics over the elements of an RDD (count, mean)
  - Save the RDD to a file (saveAsTextFile)
  - Reduce elements of an RDD into a single object or value (reduce)
- **Allows you to define partitioning/caching behavior** after defining the RDD but **before** calculating its contents.
RDD Lineage

An RDD can depend on zero or more other RDDs
● ie when $x = y.map(...)$, $x$ will depend on $y$
● These dependency relationships can be thought of as a graph.

You can call this graph a lineage graph, as it represents the derivation of each RDD
● It is also necessarily a **DAG**, since a loop is impossible to be present in it.
● Narrow dependencies, where a shuffle is not required (think map and filter) can be collapsed into a single **stage**.
  □ A stage is a unit of execution, created by the scheduler from RDD dependency graph
  □ Stages also depend on each other and the scheduler builds and uses this dependency graph (which is also necessarily a DAG) to schedule the stages
RDD Lineage

1. lines
   - filter(_.startsWith("ERROR"))
   - errors
     - filter(_.contains("HDFS"))
     - HDFS errors
       - map(_.split('t')(3))
       - time fields
**RDD Objects**

- `rdd1.join(rdd2)`
- `groupBy(...)`
- `filter(...)`

**DAGScheduler**

- DAG
- split graph into stages of tasks
- submit each stage as ready

**TaskScheduler**

- TaskSet
- launch tasks via cluster manager
- retry failed or straggling tasks

**Worker**

- Threads
- Block manager
- execute tasks
- store and serve blocks

**Tips**

- Agnostic to operators!
- Doesn’t know about stages
- Stage failed
Representing RDDs

Each RDD is represented through a common interface that exposes 5 pieces of information:

1. A set of partitions, atomic pieces of datasets
2. Set of dependencies on the parent RDDs
3. Function for computing the RDD from the parents
4. Metadata about partitioning scheme
5. Data placement

See table 3 in the RDD paper.
Dependencies

**Narrow dependencies:** each parent RDD partition used by at most one child; ie map()
- allow pipelined execution: example map() and filter() in iterative fashion
- recovery after node failure is more efficient

**Wide dependencies:** multiple child partitions may depend on a parent RDD; ie join()
- Single failed node in a wide dependency lineage graph may cause loss of partition in many ancestral dependencies