Advanced Analytics with Spark SQL and MLLib

Michael Armbrust
@michaelarmbrust

DATABRICKS

Slides available here

spark.apache.org
What is Apache Spark?

Fast and general cluster computing system interoperable with Hadoop

Improves efficiency through:
  » In-memory computing primitives
  » General computation graphs

Improves usability through:
  » Rich APIs in Scala, Java, Python
  » Interactive shell

Up to 100× faster (2-10× on disk)

2-5× less code
A Unified Stack

Spark SQL
Spark Streaming real-time
GraphX graph
MLlib machine learning

Spark
Why a New Programming Model?

MapReduce greatly simplified big data analysis

But once started, users wanted more:
  » More complex, multi-pass analytics (e.g. ML, graph)
  » More interactive ad-hoc queries
  » More real-time stream processing

All 3 need faster data sharing in parallel apps
Data Sharing in MapReduce

Slow due to replication, serialization, and disk IO
What We’d Like

Input

iter. 1

iter. 2

... one-time processing

Input

Distributed memory

query 1

query 2

query 3

... 10-100x faster than network and disk
Spark Model

*Write programs in terms of transformations on distributed datasets*

Resilient Distributed Datasets (RDDs)
- Collections of objects that can be stored in memory or disk across a cluster
- Built via parallel transformations (map, filter, …)
- Automatically rebuilt on failure
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
codes

lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda x: x.startswith("ERROR"))
messages = errors.map(lambda x: x.split(\'t\')[2])
messages.cache()

messages.filter(lambda x: "foo" in x).count()
messages.filter(lambda x: "bar" in x).count()  
...
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)
Fault Tolerance

RDDs track lineage info to rebuild lost data

```python
file.map(lambda word: (word, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (word, count): count > 1000)
```
Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```
file.map(lambda word: (word, 1))
  .reduceByKey(lambda x, y: x + y)
  .filter(lambda (word, count): count > 1000)
```
Example: Logistic Regression

**Graph:**
- **Y-axis:** Running Time (s)
- **X-axis:** Number of Iterations

- **Hadoop**
  - 110 s / iteration
  - First iteration: 80 s
  - Further iterations: 1 s

- **Spark**
Behavior with Less RAM

![Bar chart showing iteration time (s) for different cache levels. The x-axis represents % of working set in memory, and the y-axis represents iteration time (s). The cache levels are Cache disabled, 25%, 50%, 75%, and Fully cached. The iteration times are 68.8, 58.1, 40.7, 29.7, and 11.5 seconds, respectively.]
# Supported Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
<td>sample</td>
</tr>
<tr>
<td>filter</td>
<td>count</td>
<td>take</td>
</tr>
<tr>
<td>groupBy</td>
<td>fold</td>
<td>first</td>
</tr>
<tr>
<td>sort</td>
<td>reduceByKey</td>
<td>partitionBy</td>
</tr>
<tr>
<td>union</td>
<td>groupByKey</td>
<td>mapWith</td>
</tr>
<tr>
<td>join</td>
<td>cogroup</td>
<td>pipe</td>
</tr>
<tr>
<td>leftOuterJoin</td>
<td>cross</td>
<td>save</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td>zip</td>
<td>...</td>
</tr>
</tbody>
</table>
Spark Community

One of the largest open source projects in big data:

Started in 2009, open sourced 2010

150+ developers contributing

30+ companies contributing

Contributors in past year
Example: Logistic Regression

Goal: find best line separating two sets of points
Import Libraries

# Start by importing some standard libraries
# and PySpark
from collections import namedtuple
from math import exp
import numpy as np
from pyspark import SparkContext
Load and Parse Data

D = 10  # Number of dimensions

# Read a batch of points from the input file
# into a NumPy matrix object.
# The data file contains lines of the form:
# <label> <x1> <x2> ... <xD>.
def readPointBatch(iterator):
    strs = list(iterator)
    matrix = np.zeros((len(strs), D + 1))
    for i in xrange(len(strs)):
        matrix[i] = np.fromstring(strs[i].replace(',', ' ').replace('
', ' '),
                              dtype=np.float32, sep=' ')  
    return [matrix]
Calculate the gradient

\[ w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i) \]

# Compute logistic regression gradient for a matrix of points

def gradient(matrix, w):
    # point labels (first column of input file)
    Y = matrix[:, 0]
    # point coordinates
    X = matrix[:, 1:]
    # For each point (x, y), compute gradient function
    g = ((1.0 / (1.0 + np.exp(-Y * X.dot(w))) - 1.0) * Y * X.T)
    return g.sum(1)

# Add up the gradients from each distributed batch

def add(x, y):
    x += y
    return x
Putting it all together

```
sc = SparkContext(appName="PythonLR")

# Initialize w to a random value
w = 2 * np.random.rand(size=D) - 1
print "Initial w: " + str(w)

points = sc.textFile(...).mapPartitions(readPointBatch).cache()

for i in range(1000):
    print "On iteration %i" % (i + 1)
    w -= points.map(lambda m: gradient(m, w)).reduce(add)

print "Final w: " + str(w)
```
Spark MLlib

- Set of common machine learning algorithms
- Growing fast: 40+ contributors!
What is MLlib?

Algorithms:

**classification**: logistic regression, linear support vector machine (SVM), naive Bayes, classification tree

**regression**: generalized linear models (GLMs), regression tree

**collaborative filtering**: alternating least squares (ALS), non-negative matrix factorization (NMF)

**clustering**: k-means

**decomposition**: singular value decomposition (SVD), principal component analysis (PCA)
K-means with MLlib

# Load and parse the data
data = sc.textFile("kmeans_data.txt")
def parseData(line):
    return array([[float(x) for x in line.split(' ')]])
parsedData = data.map(parseData).cache()

# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations = 10,
                         runs = 1, initialization_mode = "kmeans||")

# Evaluate clustering by computing the sum of squared errors
def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

cost = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Sum of squared error = " + str(cost))
What’s new in Mlib 1.0

• new user guide and code examples
• API stability
• sparse data support
• More algorithms:
  • regression and classification tree
  • distributed matrices
  • tall-and-skinny PCA and SVD
  • L-BFGS
  • binary classification model evaluation
MLlib user guide

http://spark.apache.org/docs/latest/mllib-guide.html

Examples

Following examples can be tested in the PySpark shell.

In the following example after loading and parsing data, we use the KMeans object to cluster the data into two clusters. The number of desired clusters is passed to the algorithm. We then compute Within Set Sum of Squared Error (WSSSE). You can reduce this error measure by increasing k. In fact the optimal k is usually one where there is an “elbow” in the WSSSE graph.

```python
from pyspark.mllib.clustering import KMeans
from numpy import array
from math import sqrt

# Load and parse the data
data = sc.textFile("data/kmeans_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.split(' ')])

# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10,
runs=10, initializationMode="random")
```
Code examples

Code examples that can be used as templates to build standalone applications are included under the “examples/” folder with sample datasets:

- binary classification (linear SVM and logistic regression)
- decision tree
- k-means
- linear regression
- naive Bayes
- tall-and-skinny PCA and SVD
- collaborative filtering
API stability

Following Spark core, MLlib is aiming for binary compatibility for all 1.x releases on stable APIs.

For changes in experimental and developer APIs, we will provide migration guide between releases.
“large-scale sparse problems”

Sparse datasets appear almost everywhere in the world of big data, where the sparsity may come from many sources, e.g.,

feature transformation:
   one-hot encoding, interaction, and bucketing,

large feature space: n-grams,

missing data: rating matrix,

low-rank structure: images and signals.
Sparsity is almost everywhere

The Netflix Prize:

number of users: 480,189
number of movies: 17,770
number of observed ratings: 100,480,507

sparsity = 1.17%
Sparse representation of features

dense : 1. 0. 0. 0. 0. 0. 3.

\[
\text{size} : 7
\]

sparse :
\[
\begin{cases}
\text{indices} : 0, 6 \\
\text{values} : 1, 3.
\end{cases}
\]
Exploiting sparsity

In Spark v1.0, MLlib adds support for sparse input data in Scala, Java, and Python.

MLlib takes advantage of sparsity in both storage and computation in

- linear methods (linear SVM, logistic regression, etc)
- naive Bayes,
- k-means,
- summary statistics.
Exploiting sparsity in k-means

Training set:

- number of examples: 12 million
- number of features: 500
- sparsity: 10%

<table>
<thead>
<tr>
<th></th>
<th>dense</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage</td>
<td>47GB</td>
<td>7GB</td>
</tr>
<tr>
<td>time</td>
<td>240s</td>
<td>58s</td>
</tr>
</tbody>
</table>

Not only did we save 40GB of storage by switching to the sparse format, but we also received a 4x speedup.
• Newest component (started in November)
• Alpha Release as part of Spark 1.0
Shark modified the Hive backend to run over Spark, but had two challenges:
  » Limited integration with Spark programs
  » Hive optimizer not designed for Spark

Spark SQL reuses the best parts of Shark:

**Borrows**
- Hive data loading
- In-memory column store

**Adds**
- RDD-aware optimizer
- Rich language interfaces
Spark SQL Components

- Catalyst Optimizer
  - Relational algebra + expressions
  - Query optimization

- Spark SQL Core
  - Execution of queries as RDDs
  - Reading in Parquet, JSON ...

- Hive Support
  - HQL, MetaStore, SerDes, UDFs
Adding Schema to RDDs

Spark + RDDs

Functional transformations on partitioned collections of opaque objects.

SQL + SchemaRDDs

Declarative transformations on partitioned collections of tuples.
Using Spark SQL

**SQLContext**

- Entry point for all SQL functionality
- Wraps/extends existing spark context

```python
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```
Example Dataset

A text file filled with people’s names and ages:

Michael, 30
Andy, 31
Justin Bieber, 19
...

# Load a text file and convert each line to a dictionary.
lines = sc.textFile("examples/.../people.txt")

parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p:{"name": p[0],"age": int(p[1])})

# Infer the schema, and register the SchemaRDD as a table
peopleTable = sqlCtx.inferSchema(people)
peopleTable.registerAsTable("people")
Querying Using SQL

# SQL can be run over SchemaRDDs that have been registered as a table.

```python
tenagers = sqlCtx.sql(""
    SELECT name FROM people WHERE age >= 13 AND age <= 19"")
```

# The results of SQL queries are RDDs and support all the normal RDD operations.
```
tenNames = teenagers.map(lambda p: "Name: " + p.name)
```
Caching Tables In-Memory

Spark SQL can cache tables using an in-memory columnar format:

• Scan only required columns
• Fewer allocated objects (less GC)
• Automatically selects best compression

`cacheTable("people")`
Parquet Compatibility

Native support for reading data in Parquet:

• Columnar storage avoids reading unneeded data.

• RDDs can be written to parquet files, preserving the schema.
Using Parquet

# SchemaRDDS can be saved as Parquet files, maintaining the # schema information.
peopleTable.saveAsParquetFile("people.parquet")

# Read in the Parquet file created above. Parquet files are # self-describing so the schema is preserved. The result of # loading a parquet file is also a SchemaRDD.
parquetFile = sqlCtx.parquetFile("people.parquet")

# Parquet files can be registered as tables used in SQL.
parquetFile.registerAsTable("parquetFile")
teenagers = sqlCtx.sql(""
    SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19"")
Hive Compatibility

Interfaces to access data and code in the Hive ecosystem:

- Support for writing queries in HQL
- Catalog info from Hive MetaStore
- Tablescan operator that uses Hive SerDes
- Wrappers for Hive UDFs, UDAFs, UDTFs
import HiveContext
hiveCtx = HiveContext(sc)

hiveCtx.hql(""
  CREATE TABLE IF NOT EXISTS src (key INT, value STRING)"")

hiveCtx.hql(""
  LOAD DATA LOCAL INPATH 'examples/.../kv1.txt' INTO TABLE src"")

# Queries can be expressed in HiveQL.
results = hiveCtx.hql("FROM src SELECT key, value").collect()
SQL and Machine Learning

```scala
val trainingDataTable = sql(""
SELECT e.action, u.age, u.latitude, u.logitude
FROM Users u
JOIN Events e ON u.userId = e.userId"")

// SQL results are RDDs so can be used directly in MLLib.
val trainingData = trainingDataTable.map { row =>
  val features = Array[Double](row(1), row(2), row(3))
  LabeledPoint(row(0), features)
}
val model = new LogisticRegressionWithSGD().run(trainingData)
```
Supports Scala Too!

```scala
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._

// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people =
    sc.textFile("examples/src/main/resources/people.txt")
    .map(_.split(","))
    .map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")
```
Supports Java Too!

```java
public class Person implements Serializable {
    private String _name;
    private int _age;
    public String getName() { return _name; }
    public void setName(String name) { _name = name; }
    public int getAge() { return _age; }
    public void setAge(int age) { _age = age; }
}

JavaSQLContext ctx = new org.apache.spark.sql.api.java.JavaSQLContext(sc)
JavaRDD<Person> people = ctx.textFile("examples/src/main/resources/people.txt").map(
    new Function<String, Person>() {
        public Person call(String line) throws Exception {
            String[] parts = line.split(",");
            Person person = new Person();
            person.setName(parts[0]);
            person.setAge(Integer.parseInt(parts[1].trim()));
            return person;
        }
    });

JavaSchemaRDD schemaPeople = sqlCtx.applySchema(people, Person.class);
```
Preview: TPC-DS Results

http://databricks.com/blog/2014/06/02/exciting-performance-improvements-on-the-horizon-for-spark-sql.html
Demo: Group Tweets by Language

Raw JSON
12,207,667 Tweets

Tweet Text

Language Model

Use `jsonFile(...)` to automatically turn the raw JSON into a table and SQL to extract the fields we want.

Use Mlib to cluster tweets by language based on features extracted from the tweet text.
Thanks to Aaron Davidson for putting this together…. 

DEMO!
Get Started

Visit spark.apache.org for videos & tutorials

Download Spark bundle for CDH

Easy to run on just your laptop

Free training talks and hands-on exercises: spark-summit.org
Conclusion

Big data analytics is evolving to include:
» More complex analytics (e.g. machine learning)
» More interactive ad-hoc queries, including SQL
» More real-time stream processing

Spark is the fast platform that unifies these apps

Join us at Spark Summit 2014!
June 30-July 2, San Francisco
Efficient Expression Evaluation

Interpreting expressions (e.g., ‘a + b’) can very expensive on the JVM:

• Virtual function calls
• Branches based on expression type
• Object creation due to primitive boxing
• Memory consumption by boxed primitive objects
Interpreting “a+b”

1. Virtual call to Add.eval()
2. Virtual call to a.eval()
3. Return boxed Int
4. Virtual call to b.eval()
5. Return boxed Int
6. Integer addition
7. Return boxed result
def generateCode(e: Expression): Tree = e match { 
  case Attribute(ordinal) =>
    q"inputRow.getInt($ordinal)"
  case Add(left, right) =>
    q"""""""""""""""""""
      {
        val leftResult = ${generateCode(left)}
        val rightResult = ${generateCode(right)}
        leftResult + rightResult
      }
    """
}
Executing “a + b”

```scala
val left: Int = inputRow.getInt(0)
val right: Int = inputRow.getInt(1)
val result: Int = left + right
resultRow.setInt(0, result)
```

- Fewer function calls
- No boxing of primitives
Performance Comparison

Evaluating 'a+a+a' One Billion Times

Milliseconds

<table>
<thead>
<tr>
<th>Method</th>
<th>Milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted Evaluation</td>
<td>35</td>
</tr>
<tr>
<td>Hand-written Code</td>
<td>5</td>
</tr>
<tr>
<td>Generated with Scala Reflection</td>
<td>5</td>
</tr>
</tbody>
</table>