Why Spark Is the Next Top (Compute) Model

@deanwampler

Copyright (c) 2014-2015, Dean Wampler, All Rights Reserved, unless otherwise noted. Image: Detail of the London Eye
“Trolling the Hadoop community since 2012...”

dean.wampler@typesafe.com
polyglotprogramming.com/talks
@deanwampler

Tuesday, October 20, 15
About me. You can find this presentation and others on Big Data and Scala at polyglotprogramming.com. Programming Scala, 2nd Edition is forthcoming.
photo: Dusk at 30,000 ft above the Central Plains of the U.S. on a Winter’s Day.
This page provides more information, as well as results of a recent survey of Spark usage, blog posts and webinars about the world of Spark.
This page provides more information, as well as results of a recent survey of Spark usage, blog posts and webinars about the world of Spark.
Hadoop

circa 2013

Tuesday, October 20, 15

The state of Hadoop as of last year.

Image: Detail of the London Eye
Schematic view of a Hadoop 2 cluster. For a more precise definition of the services and what they do, see e.g., http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html. We aren't interested in great details at this point, but we'll call out a few useful things to know.
Hadoop 2 uses YARN to manage resources via the master Resource Manager, which includes a pluggable job scheduler and an Applications Manager. It coordinates with the Node Manager on each node to schedule jobs and provide resources. Other services involved, including application-specific Containers and Application Masters are not shown.
Hadoop 2 clusters federate the Name node services that manage the file system, HDFS. They provide horizontal scalability of file-system operations and resiliency when service instances fail. The data node services manage individual blocks for files.
MapReduce

The classic compute model for Hadoop

Hadoop 2 clusters federate the Name node services that manage the file system, HDFS. They provide horizontal scalability of file-system operations and resiliency when service instances fail. The data node services manage individual blocks for files.
You get 1 map step (although there is limited support for chaining mappers) and 1 reduce step. If you can’t implement an algorithm in these two steps, you can chain jobs together, but you’ll pay a tax of flushing the entire data set to disk between these jobs.
The inverted index is a classic algorithm needed for building search engines.
Before running MapReduce, crawl the interwebs, find all the pages, and build a data set of URLs -> doc contents, written to flat files in HDFS or one of the more “sophisticated” formats.
Now we’re running MapReduce. In the map step, a task (JVM process) per file block (64MB or larger) reads the rows, tokenizes the text and outputs key-value pairs ("tuples").
... the keys are each word found and the values are themselves tuples, each URL and the count of the word. In our simplified example, there are typically only single occurrences of each word in each document. The real occurrences are interesting because if a word is mentioned a lot in a document, the chances are higher that you would want to find that document in a search for that word.
The output tuples are sorted by key locally in each map task, then “shuffled” over the cluster network to reduce tasks (each a JVM process, too), where we want all occurrences of a given key to land on the same reduce task.
Finally, each reducer just aggregates all the values it receives for each key, then writes out new files to HDFS with the words and a list of (URL-count) tuples (pairs).
Finally, each reducer just aggregates all the values it receives for each key, then writes out new files to HDFS with the words and a list of (URL-count) tuples (pairs).
What's not to like?

This seems okay, right? What's wrong with it?
Restrictive model makes most algorithms hard to implement.

Awkward

Writing MapReduce jobs requires arcane, specialized skills that few master. For a good overview, see http://lintool.github.io/MapReduceAlgorithms/.
The inflexible compute model leads to complex code to improve performance where hacks are used to work around the limitations. Hence, optimizations are hard to implement. The Spark team has commented on this, see http://databricks.com/blog/2014/03/26/Spark-SQL-manipulating-structured-data-using-Spark.html
Performance

Full dump of intermediate data to disk between jobs.

Sequencing jobs wouldn’t be so bad if the “system” was smart enough to cache data in memory. Instead, each job dumps everything to disk, then the next job reads it back in again. This makes iterative algorithms particularly painful.
Streaming

MapReduce only supports “batch mode”

Processing data streams as soon as possible has become very important. MR can’t do this, due to its coarse-grained nature and relative inefficiency, so alternatives have to be used.
Enter Spark
spark.apache.org
Cluster Computing
Can be run in:
• YARN (Hadoop 2)
• Mesos (Cluster management)
• EC2
• Standalone mode
• Cassandra, Riak, ...
• ...

If you have a Hadoop cluster, you can run Spark as a seamless compute engine on YARN. (You can also use pre-YARN Hadoop v1 clusters, but there you have manually allocate resources to the embedded Spark cluster vs Hadoop.) Mesos is a general-purpose cluster resource manager that can also be used to manage Hadoop resources. Scripts for running a Spark cluster in EC2 are available. Standalone just means you run Spark’s built-in support for clustering (or run locally on a single box - e.g., for development). EC2 deployments are usually standalone... Finally, database vendors like Datastax are integrating Spark.
Once you learn the core set of primitives, it’s easy to compose non-trivial algorithms with little code.
RDDs shard the data over a cluster, like a virtualized, distributed collection (analogous to HDFS). They support intelligent caching, which means no naive flushes of massive datasets to disk. This feature alone allows Spark jobs to run 10-100x faster than comparable MapReduce jobs! The “resilient” part means they will reconstitute shards lost due to process/server crashes.
RDDs shard the data over a cluster, like a virtualized, distributed collection (analogous to HDFS). They support intelligent caching, which means no naive flushes of massive datasets to disk. This feature alone allows Spark jobs to run 10-100x faster than comparable MapReduce jobs! The “resilient” part means they will reconstitute shards lost due to process/server crashes.
Once you learn the core set of primitives, it’s easy to compose non-trivial algorithms with little code.
Let’s see an actual implementation of the inverted index. First, a Hadoop MapReduce (Java) version, adapted from https://developer.yahoo.com/hadoop/tutorial/module4.html#solution It’s about 90 lines of code, but I reformatted to fit better. This is also a slightly simpler version than the one I diagrammed. It doesn’t record a count of each word in a document; it just writes (word,doc-title) pairs out of the mappers and the final (word,list) output by the reducers just has a list of documents, hence repeats. A second job would be necessary to count the repeats.
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class LineIndexer {

public static void main(String[] args) {
    JobClient client = new JobClient();
    JobConf conf =
        new JobConf(LineIndexer.class);
    conf.
        setJobName("LineIndexer");
    conf.
        setOutputKeyClass(Text.class);
    conf.
        setOutputValueClass(Text.class);
    FileInputFormat.
        addInputPath(conf,
            new Path("input"));
    FileOutputFormat.
        setOutputPath(conf,
            new Path("output"));
    conf.
        setMapperClass(LineIndexMapper.class);
    conf.
        setReducerClass(LineIndexReducer.class);
    client.
        setConf(conf);
    try {
        JobClient.
            runJob(conf);
    } catch (Exception e) {
        e.
            printStackTrace();
    }
}

public static class LineIndexMapper
extends MapReduceBase
implements Mapper<LongWritable, Text,
Text, Text> {

private final static Text word =
    new Text();
private final static Text location =
    new Text();

public void map( 
    LongWritable key, Text val,
    OutputCollector<Text, Text> output,
    Reporter reporter) throws IOException {
    FileSplit fileSplit =
        (FileSplit)reporter.
            getInputSplit();
    String fileName =
        fileSplit.
            getPath().
            getName();
    location.
        set(fileName);
    String line = val.
        toString();
    StringTokenizer itr = new
        StringTokenizer(line.
            toLowerCase());
    while (itr.
        hasMoreTokens()) {
        word.
            set(itr.
                nextToken());
        output.
            collect(word, location);
    }
}

public static class LineIndexReducer
extends MapReduceBase
implements Reducer<Text, Text,
Text, Text> {

public void reduce( 
    Text key, 
    Iterator<Text> values,
    OutputCollector<Text, Text> output,
    Reporter reporter) throws IOException {
    boolean first = true;
    StringBuilder toReturn =
        new StringBuilder();
    while (values.
        hasNext()) {
        if (!first)
            toReturn.
                append(
                    ", ");
        first=false;
        toReturn.
            append(values.
                next().
                toString());
    }
    output.
        collect(key,
            new Text(toReturn.
                toString()));
}
}

I’m not going to explain this in much detail. I used yellow for method calls, because methods do the real work!! But notice that the functions in this code don’t really do a whole lot, so there’s low information density and you do a lot of bit twiddling.
import java.io.IOException;
import java.util.*
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.
import org.apache.hadoop.mapred.*

public class LineIndexer {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf =
            new JobConf(LineIndexer.class);
        conf.setJobName("LineIndexer");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(conf,
            new Path("input"));
        FileOutputFormat.setOutputPath(conf,
            new Path("output"));
        conf.setMapperClass(
            LineIndexMapper.class);
        conf.setReducerClass(
            LineIndexReducer.class);
        client.setConf(conf);

        try {
            JobClient.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static class LineIndexMapper
        extends MapReduceBase
        implements Mapper<LongWritable, Text,
            Text, Text> {
        private final static Text word =
            new Text();
        private final static Text location =
            new Text();
        public void map(
            LongWritable key, Text val,
            OutputCollector<Text, Text> output,
            Reporter reporter) throws IOException {
            FileSplit fileSplit =
                (FileSplit)reporter.
                    getInputSplit();
            String fileName =
                fileSplit.
                    getPath().
                    getName();
            location.
                set(fileName);
            String line = val.
                toString();
            StringTokenizer itr = new
                StringTokenizer(line.
                    toLowerCase());
            while (itr.
                hasMoreTokens()) {
                word.
                    set(itr.
                        nextToken());
                output.
                    collect(word, location);
            }
        }

        public static class LineIndexReducer
            extends MapReduceBase
            implements Reducer<Text, Text,
                Text, Text> {
        public void reduce(
            Text key,
            Iterator<Text> values,
            OutputCollector<Text, Text> output,
            Reporter reporter) throws IOException {
            boolean first = true;
            StringBuilder toReturn =
                new StringBuilder();
            while (values.
                hasNext()) {
                if (!first)
                    toReturn.
                        append(
                            ",");
                first=false;
                toReturn.
                    append(1
                        values.
                        next().
                            toString());
            }
            output.
                collect(key,
                    new Text(toReturn.
                        toString()));
        }
    }
}
import java.io.IOException;
import java.util.*
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*
import org.apache.hadoop.mapred.*

public class LineIndexer {

public static void main(String[] args) {

JobClient client = new JobClient();
JobConf conf =
new JobConf(LineIndexer.class);
conf.setJobName("LineIndexer");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
FileInputFormat.addInputPath(conf, new Path("input"));
FileOutputFormat.setOutputPath(conf, new Path("output"));
conf.setMapperClass(LineIndexMapper.class);
conf.setReducerClass(LineIndexReducer.class);
client.setConf(conf);
try {
    JobClient.runJob(conf);
} catch (Exception e) {
    e.printStackTrace();
}
}

public static class LineIndexMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {
private final static Text word =
new Text();
private final static Text location =
new Text();
public void map(LongWritable key, Text val,
OutputCollector<Text, Text> output,
Reporter reporter) throws IOException {
    FileSplit fileSplit =
(reporter.getInputSplit());
    String fileName =
(fileSplit.getPath().getName());
    location.set(fileName);
    String line = val.toString();
    StringTokenizer itr = new
(StringTokenizer(line.toLowerCase()));
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        output.collect(word, location);
    }
}

public static class LineIndexReducer extends MapReduceBase implements Reducer<Text, Text, Text, Text> {
public void reduce(Text key,
Iterator<Text> values,
OutputCollector<Text, Text> output,
Reporter reporter) throws IOException {
    boolean first = true;
    StringBuilder toReturn =
(new StringBuilder());
    while (values.hasNext()) {
        if (!first)
            toReturn.append(", ");
        first=false;
        toReturn.append(values.next().toString());
    }
    output.collect(key,
(new Text(toReturn.toString())));
}
}

public static class LineIndexMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {

}

main ends with a try-catch clause to run the job.
This is the LineIndexMapper class for the mapper. The map method does the real work of tokenization and writing the (word, document-name) tuples.
FilesSplit fileSplit =
    (FilesSplit)reporter.getInputSplit();
String fileName =
    fileSplit.getPath().getName();
location.set(fileName);

String line = val.toString();
StringTokenizer itr = new
    StringTokenizer(line.toLowerCase());
while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    output.collect(word, location);
}

Tuesday, October 20, 15
The rest of the LineIndexMapper class and map method.
The reducer class, LineIndexReducer, with the reduce method that is called for each key and a list of values for that key. The reducer is stupid; it just reformats the values collection into a long string and writes the final (word,list-string) output.
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class LineIndexer {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf = new JobConf(LineIndexer.class);
        conf.setJobName("LineIndexer");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(conf, new Path("input"));
        FileOutputFormat.setOutputPath(conf, new Path("output"));
        conf.setMapperClass(LineIndexMapper.class);
        conf.setReducerClass(LineIndexReducer.class);
        client.setConf(conf);
        try {
            JobClient.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static class LineIndexMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {
        private final static Text word = new Text();
        private final static Text location = new Text();
        public void map(LongWritable key, Text val, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
            FileSplit fileSplit = (FileSplit)reporter.getInputSplit();
            String fileName = fileSplit.getPath().getName();
            location.set(fileName);
            String line = val.toString();
            StringTokenizer itr = new StringTokenizer(line.toLowerCase());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                output.collect(word, location);
            }
        }
    }

    public static class LineIndexReducer extends MapReduceBase implements Reducer<Text, Text, Text, Text> {
        public void reduce(Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
            boolean first = true;
            StringBuilder toReturn = new StringBuilder();
            while (values.hasNext()) {
                if (!first)
                    toReturn.append("", ");
                first=false;
                toReturn.append(values.next().toString());
            }
            output.collect(key, new Text(toReturn.toString()));
        }
    }
}
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class LineIndexer {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf =
            new JobConf(LineIndexer.class);
        conf.setJobName("LineIndexer");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(conf, new Path("input"));
        FileOutputFormat.setOutputPath(conf, new Path("output"));
        conf.setMapperClass(LineIndexMapper.class);
        conf.setReducerClass(LineIndexReducer.class);
        client.setConf(conf);
        try {
            client.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static class LineIndexMapper
    extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, Text> {
    private final static Text word =
        new Text();
    private final static Text location =
        new Text();
    public void map(
        LongWritable key, Text val,
        OutputCollector<Text, Text> output,
        Reporter reporter) throws IOException {
        FileSplit fileSplit =
            (FileSplit)reporter.getInputSplit();
        String fileName =
            fileSplit.getPath().getName();
        location.set(fileName);
        String line = val.toString();
        StringTokenizer itr = new
            StringTokenizer(line.toLowerCase());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, location);
        }
    }
}

public static class LineIndexReducer
    extends MapReduceBase
    implements Reducer<Text, Text, Text, Text> {
    public void reduce(
        Text key, Iterator<Text> values,
        OutputCollector<Text, Text> output,
        Reporter reporter) throws IOException {
        boolean first = true;
        StringBuilder toReturn =
            new StringBuilder();
        while (values.hasNext()) {
            if (!first) {
                toReturn.append(', ');
                first = false;
            }
            toReturn.append(values.next());
        }
        output.collect(key, new Text(toReturn.toString()));
    }
}
Inverted Index in Spark (Scala).

This code is approximately 45 lines, but it does more than the previous Java example, it implements the original inverted index algorithm I diagrammed where word counts are computed and included in the data.
The InvertedIndex implemented in Spark. This time, we’ll also count the occurrences in each document (as I originally described the algorithm) and sort the (url,N) pairs descending by N (count), and ascending by URL.
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
  def main(a: Array[String]) = {
    val sc = new SparkContext(
      "local[*]", "Inverted Idx")
    .textFile("data/crawl")
    .map(line =>
      val Array(path, text) =
        line.split("\t",2)
      (path, text)
    )
    .flatMap{
      case (path, text) =>
        text.split("\W+")
        .map{
          word =>
            (word, path)
        }
    }
    .map{
      case (w, p) =>
        ((w, p), 1)
    }
    .reduceByKey{
      case (n1, n2) =>
        n1 + n2
    }
    .map{
      case ((w, p), n) =>
        (w, (p, n))
    }
    .groupByKey
    .mapValues{
      iter =>
        iter.toSeq
        .sortBy{
          case (path, n) =>
            (-n, path)
        }
        .mkString(", ")
    }
    .saveAsTextFile("/path/out")
    sc.stop()
  }
}
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
  def main(a: Array[String]) = {
    val sc = new SparkContext( "local[*]", "Inverted Idx"
    .textFile("data/crawl")
    .map(line =>
      val Array(path, text) =
        line.split("\t", 2)
    )
    .flatMap{
      case (path, text) =>
        text.split("\W+")
      .map{
        word =>
          (word, path)
      }
    }
    .map{
      case (w, p) =>
        ((w, p), 1)
    }
    .reduceByKey{
      case (n1, n2) =>
        n1 + n2
    }
    .map{
      case ((w, p), n) =>
        (w, (p, n))
    }
    .groupByKey
    .mapValues{
      iter =>
        iter.toSeq
      .sortBy{
        case (path, n) =>
          (-n, path)
      }
      .mkString(", ")
    }
    .saveAsTextFile("/path/out")
    sc.stop()
  }
}

You being the workflow by declaring a SparkContext. We’re running in “local[*]” mode, in this case, meaning on a single machine, but using all cores available. Normally this argument would be a command-line parameter, so you can develop locally, then submit to a cluster, where “local” would be replaced by the appropriate cluster master URI.
The rest of the program is a sequence of function calls, analogous to “pipes” we connect together to construct the data flow. Data will only start “flowing” when we ask for results.

We start by reading one or more text files from the directory “data/crawl”. If running in Hadoop, if there are one or more Hadoop-style “part-NNNNN” files, Spark will process all of them (they will be processed synchronously in “local” mode).
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
  def main(a: Array[String]) = {
    val sc = new SparkContext("local[*]", "Inverted Idx")
    sc.textFile("data/crawl")
      .map({ line =>
        val Array(path, text) = line.split("\t", 2)
        (path, text)
      })
      .flatMap{
        case (path, text) =>
          text.split("\W+") map {
            word => (word, path)
          }
      }
      .map{
        case ((w, p), n) => ((w, p), 1)
      }
      .reduceByKey{
        case (n1, n2) => n1 + n2
      }
      .map{
        case ((w, p), n) => (w, (p, n))
      }
      .groupByKey
      .mapValues{
        iter =>
          iter.toSeq.sortBy{
            case (path, n) => (-n, path)
          }.mkString(",
        )
      }
      .saveAsTextFile("/path/out")
    sc.stop()
  }
}

sc.textFile returns an RDD with a string for each line of input text. So, the first thing we do is map over these strings to extract the original document id (i.e., file name), followed by the text in the document, all on one line. We assume tab is the separator. “(array(0), array(1))” returns a two-element “tuple”. Think of the output RDD has having a schema “fileName: String, text: String”.
sc.textFile("data/crawl")
.map { line =>
  val Array(path, text) =
  line.split("\t",2)
  (path, text)
}
.flatMap {
  case (path, text) =>
  text.split(""\W+""") map {
    word => (word, path)
  }
}
.map {
  case ((w, p), n) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupByKey
.mapValues {
  iter =>
  iter.toSeq.sortBy {
    case (path, n) => (-n, path)
  } mkString ", "
}
.saveAsTextFile("/path/out")
sc.stop()
Next, we map over these pairs and add a single "seed" count of 1. Note the structure of the returned tuple; it’s a two-tuple where the first element is itself a two-tuple holding (word, path). The following special method, reduceByKey is like a groupBy, where it groups over those (word, path) "keys" and uses the function to sum the integers. The popup shows the what the output data looks like.
So, the input to the next map is now ((word, path), n), where n is now >= 1. We transform these tuples into the form we actually want, (word, (path, n)). I love how concise and elegant this code is!
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
  def main(a: Array[String]) = {
    val sc = new SparkContext("local[*]", "Inverted Idx")
    sc.textFile("data/crawl").map{
      line =>
        val Array(path, text) = line.split("\t",2)
    }.
    flatMap{
      case (path, text) =>
        text.split("\W+").map{
          word => (word, path)
        }
    }.
    map{
      case (w, p) => ((w, p), 1)
    }.
    reduceByKey{
      case (n1, n2) => n1 + n2
    }.
    map{
      case ((w, p), n) => (w, (p, n))
    }.
    groupByKey.
    mapValues{
      iter =>
        iter.toSeq.sortBy{
          case (path, n) => (-n, path)
        }.mkString(", ")
    }.
    saveAsTextFile("/path/out")
    sc.stop()}
  }

Now we do an explicit group by to bring all the same words together. The output will be (word, seq((path1, n1), (path2, n2), ...)).
The last map over just the values (keeping the same keys) sorts by the count descending and path ascending. (Sorting by path is mostly useful for reproducibility, e.g., in tests!).
Finally, write back to the file system and stop the job.
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
  def main(a: Array[String]) = {
    val sc = new SparkContext("local[*]", "Inverted Idx")
    sc.textFile("data/crawl")
      .map { line =>
        val Array(path, text) = line.split("\t", 2)
        (path, text)
      }
      .flatMap {
        case (path, text) =>
          text.split("\W+")
            .map {
              word => (word, path)
            }
      }
      .map {
        case (w, p) => ((w, p), 1)
      }
      .reduceByKey {
        case (n1, n2) => n1 + n2
      }
      .map {
        case ((w, p), n) => (w, (p, n))
      }
      .groupByKey
      .mapValues { iter =>
        iter.toSeq.sortBy {
          case (path, n) => (-n, path)
        }
      }
      .saveAsTextFile("/path/to/out")
    sc.stop()
  }
}

Altogether
Concise Operators!

Once you have this arsenal of concise combinators (operators), you can compose sophisticated dataflows very quickly.
Another example of a beautiful and profound DSL, in this case from the world of Physics: Maxwell’s equations: http://upload.wikimedia.org/wikipedia/commons/c/c4/Maxwell'sEquations.svg
Once you learn the core primitives I used, and a few tricks for manipulating the RDD tuples, you can very quickly build complex algorithms for data processing!

The Spark API allowed us to focus almost exclusively on the “domain” of data transformations, while the Java MapReduce version (which does less), forced tedious attention to infrastructure mechanics.

The Spark version took me ~30 minutes to write.
Use a SQL query when you can!!
Spark SQL!

New DataFrame API with query optimizer (equal performance for Scala, Java, Python, and R), Python/R-like idioms.

This API sits on top of a new query optimizer called Catalyst, that supports equally fast execution for all high-level languages, a first in the big data world.
Spark SQL!

Mix SQL queries with the RDD API.
Interoperate with Hive, the original Hadoop SQL tool.
Read strings that are JSON records, infer the schema on the fly. Also, write RDD records as JSON.
Parquet is a newer file format developed by Twitter and Cloudera that is becoming very popular. It stores in column order, which is better than row order when you have lots of columns and your queries only need a few of them. Also, columns of the same data types are easier to compress, which Parquet does for you. Finally, Parquet files carry the data schema.
SparkSQL

~10-100x the performance of Hive, due to in-memory caching of RDDs & better Spark abstractions.
Combine SparkSQL queries with Machine Learning code.

We’ll use the Spark “MLlib” in the example, then return to it in a moment.
CREATE TABLE Users(
  userId INT,
  name STRING,
  email STRING,
  age INT,
  latitude DOUBLE,
  longitude DOUBLE,
  subscribed BOOLEAN);

CREATE TABLE Events(
  userId INT,
  action INT);

This example adapted from the following blog post announcing Spark SQL:

Adapted here to use Spark’s own SQL, not the integration with Hive. Imagine we have a stream of events from users and the events that have occurred as they used a system.
val trainingDataTable = sql(""
SELECT e.action, u.age,
    u.latitude, u.longitude
FROM Users u
JOIN Events e
ON u.userId = e.userId"")

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)

val allCandidates = sql(""
SELECT userId, age, latitude, longitude
FROM Users
WHERE subscribed = FALSE"")

case class Score(
    userId: Int, score: Double)
val scores = allCandidates
    map { row =>
        val features =
            Array[Double](row(1), row(2), row(3))
        Score(row(0), model.
            predict(features))
    }

// In-memory table
scores.
    registerTempTable("Scores")

val topCandidates = sql(""
SELECT u.name, u.email
FROM Scores s
JOIN Users u ON s.userId = u.userId
ORDER BY score DESC
LIMIT 100"")

Here is some Spark (Scala) code with an embedded SQL query that joins the Users and Events tables. The """"..."""" string allows embedded line feeds.
The "sql" function returns an RDD. If we used the Hive integration and this was a query against a Hive table, we would use the hql(...) function instead.
val trainingDataTable = sql(""
SELECT e.action, u.age,
    u.latitude, u.longitude
FROM Users u
JOIN Events e
ON u.userId = e.userId"")

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)

val allCandidates = sql(""
SELECT userId, age, latitude, longitude
FROM Users
WHERE subscribed = FALSE"")

case class Score(userId: Int, score: Double)

val scores = allCandidates map {
    row =>
    val features = Array[Double](row(1), row(2), row(3))
    Score(row(0), model.predict(features))
}

// In-memory table
scores.registerTempTable("Scores")

val topCandidates = sql(""
SELECT u.name, u.email
FROM Scores s
JOIN Users u ON s.userId = u.userId
ORDER BY score DESC
LIMIT 100"")
Next we train the recommendation engine, using a “logistic regression” fit to the training data, where “stochastic gradient descent” (SGD) is used to train it. (This is a standard tool set for recommendation engines; see for example: http://www.cs.cmu.edu/~wcohen/10-605/assignments/sgd.pdf)
val model = 
  new LogisticRegressionWithSGD().
  .run(trainingData)

val allCandidates = sql(""
  SELECT userId, age, latitude, longitude
  FROM Users
  WHERE subscribed = FALSE"")

case class Score(
  userId: Int, score: Double)
val scores = allCandidates map { row => 
  val features =
    Array[Double](row(1), row(2), row(3))
  Score(row(0), model.predict(features))
}

Now run a query against all users who aren’t already subscribed to notifications.
case class Score(
    userId: Int, score: Double)
val scores = allCandidates map { row =>
    val features =
        Array[Double](row(1), row(2), row(3))
    Score(row(0), model.predict(features))
}

// In-memory table
scores.registerTempTable("Scores")

val topCandidates = sql(""")
    SELECT u.name, u.email
    FROM Scores s
    JOIN Users u ON s.userId = u.userId
    ORDER BY score DESC
case class Score(
    userId: Int, score: Double)
val scores = allCandidates map { row =>
    val features = Array[Double](row(1), row(2), row(3))
    Score(row(0), model.predict(features))
}

// In-memory table
scores.registerTempTable("Scores")

val topCandidates = sql(""")
    SELECT u.name, u.email
    FROM Scores s
    JOIN Users u ON s.userId = u.userId
    ORDER BY score DESC

Then “register” the scores RDD as a “Scores” table in in memory. If you use the Hive binding instead, this would be a table in Hive’s metadata storage.
// In-memory table
scores.registerTempTable("Scores")

val topCandidates = sql(""
SELECT u.name, u.email
FROM Scores s
JOIN Users u ON s.userId = u.userId
ORDER BY score DESC
LIMIT 100"")

Finally, run a new query to find the people with the highest scores that aren’t already subscribing to notifications. You might send them an email next recommending that they subscribe...
val trainingDataTable = sql(""")
  SELECT e.action, u.age,
           u.latitude, u.longitude
  FROM Users u
  JOIN Events e
  ON u.userId = e.userId"")

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)

val allCandidates = sql(""")
  SELECT userId, age, latitude, longitude
  FROM Users
  WHERE subscribed = FALSE"")

case class Score(
  userId: Int, score: Double)
val scores = allCandidates map { row =>
  val features =
    Array[Double](row(1), row(2), row(3))
  Score(row(0), model.predict(features))
}

// In-memory table
scores.registerTempTable("Scores")

val topCandidates = sql(""")
  SELECT u.name, u.email
  FROM Scores s
  JOIN Users u ON s.userId = u.userId
  ORDER BY score DESC
  LIMIT 100"")
Event Stream Processing
Use the same abstractions for near real-time, event streaming.
A DStream (discretized stream) wraps the RDDs for each “batch” of events. You can specify the granularity, such as all events in 1 second batches, then your Spark job is passed each batch of data for processing. You can also work with moving windows of batches.
Very similar code...
val sc  = new SparkContext(...)  
val ssc = new StreamingContext(  
    sc, Seconds(10))  

// A DStream that will listen  
// for text on server:port  
val lines =  
    ssc.socketTextStream(s, p)  

// Word Count...  
val words = lines flatMap {  
    line => line.split("\W+")  
}
val sc = new SparkContext(...) 
val ssc = new StreamingContext(
  sc, Seconds(10))

// A DStream that will listen
// for text on server:port
val lines = 
  ssc.socketTextStream(s, p)

// Word Count...
val words = lines flatMap 
  { line => line.split("\W+") }

We create a StreamingContext that wraps a SparkContext (there are alternative ways to construct it...). It will “clump” the events into 1-second intervals.
val sc = new SparkContext(...)
val ssc = new StreamingContext(
    sc, Seconds(10))

// A DStream that will listen
// for text on server:port
val lines =
    ssc.socketTextStream(s, p)

// Word Count...
val words = lines flatMap {
    line => line.split("\W+")
}

wordCount.
    saveAsTextFiles(outpath)
ssc.
    start
ssc.
    awaitTermination

Next we setup a socket to stream text to us from another server and port (one of several ways to ingest data).
// Word Count...
val words = lines flatMap {
  line => line.split("\"\"\"\W+"\"\"\")
}

val pairs = words map ((_, 1))
val wordCounts =
pairs reduceByKey ((i,j) => i+j)
wordCount.saveAsTextFiles(outpath)
ssc.start()
/ Word Count...
val words = lines flatMap {
  line => line.split("\W+")
}

val pairs = words map ((_, 1))
val wordCounts =
pairs reduceByKey ((i,j) => i+j)
wordCount.saveAsTextFiles(outpath)
ssc.start()
val pairs = words map ((_, 1))
val wordCounts =
pairs reduceByKey ((i,j) => i+j)

wordCount.saveAsTextFiles(outpath)

ssc.start()
ssc.awaitTermination()
val pairs = words map ((_, 1))
val wordCounts =
  pairs reduceByKey ((i,j) => i+j)
wordCount.saveAsTextFiles(outpath)

ssc.start()
ssc.awaitTermination()
Machine Learning Library...

MLlib, which we won’t discuss further.
Distributed Graph Computing...

GraphX, which we won't discuss further.
Some problems are more naturally represented as graphs.
Extends RDDs to support property graphs with directed edges.
Spark

A flexible, scalable distributed compute platform with concise, powerful APIs and higher-order tools.

spark.apache.org
Image: The London Eye on one side of the Thames, Parliament on the other.