CinJUG 11/13/2014
ScalaDays 2014

Typesafe

Why Scala Is Taking Over the Big Data World

SOME RIGHTS RESERVED

Sydney Opera House photos Copyright © Dean Wampler, 2011-2014, Some Rights Reserved.
The content is free to reuse, but attribution is requested.
http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode

Dean Wampler

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

About me. You can find this presentation and others on Big Data and Scala at polyglotprogramming.com.

Thursday, November 13, 14

Let's put all this into perspective...

http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg

Thursday, November 13, 14

Let's put all this into perspective, circa 2013...

http://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Whole_world_-_land_and_oceans_12000.jpg/1280px-Whole_world_-_land_and_oceans_12000.jpg

"Dataists" - Data Architects/DBAs: SQL, SAS. "DBs" could also be distributed file systems.
Data Scientists - Statistics experts. Some programming, especially Python, R, Julia, maybe Matlab, etc.
Developers like us, who figure out the infrastructure (but don't usually manage it), and write the programs that do batch-oriented ETL (extract, transform, and load), and more real-time event handling.
Often this data gets pumped into Hadoop or other compute engines and data stores, including various SQL and NoSQL DBs, and file systems.

What we'll focus on.

Let's drill down to Hadoop, circa 2013.

The schematic view of a Hadoop v2 cluster, with federated master services for job and other resource management, and the file system, plus slave nodes to run individual compute tasks in parallel (each a JVM process) and store blocks of the files (with 3x block replication for resiliency).

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

Historically, up to 2013, MapReduce was the officially-supported compute engine for writing all compute jobs.

1 Map step + 1 Reduce step

Problems

Hard to implement algorithms...

Nontrivial algorithms are hard to convert to just map and reduce steps, even though you can sequence multiple map+reduce “jobs”. It takes specialized expertise of the tricks of the trade.

Problems

... and the Hadoop API is horrible:

The Hadoop API is very low level and tedious...

```
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);
```


For example, the classic inverted index, used to convert an index of document locations (e.g., URLs) to words into the reverse; an index from words to doc locations. It's the basis of search engines.

I'm not going to explain the details. The point is to notice all the boilerplate that obscures the problem logic.

Everything is in one outer class. We start with a main routine that sets up the job.

I used yellow for method calls, because methods do the real work!! But notice that most of the functions in this code don't really do a whole lot of work for us...

```
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);
```


boilerplate...

```
    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> {
```


main ends with a try-catch clause to run the job.

```
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 =
```


This is the LineIndexMapper class for the mapper. The map method does the real work of tokenization and writing the (word, document-name) tuples.

```
    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);
    }
  }
}
```


The rest of the LineIndexMapper class and map method.

```
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());
```


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.

```
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()));
    }
  }
}
```


EOF

Altogether

```
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()));
  }
 }
}
```


The whole shebang (6pt. font) This would take a few hours to write, test, etc. assuming you already know the API and the idioms for using it.

Solution

Use Scalding

Twitter wrote a Scala API, https://github.com/twitter/scalding, to hide the mess. Actually, Scalding sits on top of Cascading (http://cascading.org) a higher-level Java API that exposes more sensible “combinators” of operations, but is still somewhat verbose due to the pre-Java 8 conventions it must use. Scalding gives us the full benefits of Scala syntax and functional operations, “combinators”.

```
import com.twitter.scalding._

class InvertedIndex(args: Args)
  extends Job(args) {

  val texts = Tsv("texts.tsv", ('id, 'text))

  val wordToIds = texts
    .flatMap(('id, 'text) -> ('word, 'id2)) {
      fields: (Long, String) =>
       val (id2, text) =
        fields.text.split("\\s+").map {
         word => (word, id2)
        }
  }
```


Dramatically smaller, succinct code! (https://github.com/echen/rosetta-scone/blob/master/inverted-index/InvertedIndex.scala) Note that this example assumes a slightly different input data format (more than one document per file, with each document id followed by the text all on a single line, tab separated.

```
val wordIds = texts
  .flatMap(('id, 'text) -> ('word, 'id2)) {
    fields: (Long, String) =>
     val (id2, text) =
       fields.text.split("\\s+").map {
        word => (word, id2)
       }
  }

val invertedIndex =
 wordToTweets.groupBy('word) {
   _.toList[Long]('id2 -> 'ids)
  }
invertedIndex.write(Tsv("output.tsv"))
}
```


Dramatically smaller, succinct code! (https://github.com/echen/rosetta-scone/blob/master/inverted-index/InvertedIndex.scala) Note that this example assumes a slightly different input data format (more than one document per file, with each document id followed by the text all on a single line, tab separated.

```
  val wordfolds = texts
    .flatMap(('id, 'text) -> ('word, 'id2)) {
     fields: (Long, String) =>
      val (id2, text) =
        fields.text.split("\\s+").map {
         word => (word, id2)
        }
    }


  val invertedIndex =
   wordToTweets.groupBy('word) {
     _.toList[Long]('id2 -> 'ids)
   }
  invertedIndex.write(Tsv("output.tsv"))
}
```

That's it!

Dramatically smaller, succinct code! (https://github.com/echen/rosetta-scone/blob/master/inverted-index/InvertedIndex.scala) Note that this example assumes a slightly different input data format (more than one document per file, with each document id followed by the text all on a single line, tab separated.

Problems

MapReduce is "batch-mode" only

You can't do event-stream processing (a.k.a. "real-time") with MapReduce, only batch mode processing.

Storm is a popular framework for scalable, resilient, event-stream processing.
Twitter wrote a Scalding-like API called Summingbird (https://github.com/twitter/summingbird) that abstracts over Storm and Scalding, so you can write one program that can run in batch mode or process events.
(For time's sake, I won't show an example.)

Problems

Flush to disk, then reread between jobs

Wasteful

While your algorithm may be implemented using a sequence of MR jobs (which takes specialized skills to write...), the runtime system doesn't understand this, so the output of each job is flushed to disk (HDFS), even if it's TBs of data. Then it is read back into memory as soon as the next job in the sequence starts!

This problem plagues Scalding (and Cascading), too, since they run on top of MapReduce (although Cascading is being ported to Spark, which we'll discuss next). However, as of mid-2014, Cascading is being ported to a new, faster runtime called Apache Tez, and it might be ported to Spark, which we'll discuss. Twitter is working on its own optimizations within Scalding. So the perf. issues should go away by the end of 2014.

The Hadoop community has realized over the last several years that a replacement for MapReduce is needed. While MR has served the community well, it's a decade old and shows clear limitations and problems, as we've seen. In late 2013, Cloudera, the largest Hadoop vendor officially embraced Spark as the replacement. Most of the other Hadoop vendors followed.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

   val sc = new SparkContext(
    "local", "Inverted Index")

   sc.textFile("data/crawl")
    .map { line =>
      val array = line.split("\t", 2)
      (array(0), array(1))
    }
    .flatMap {
      case (path, text) =>
        text.split("""\W+""").map {
```


This implementation is more sophisticated than the Scalding example. It also computes the count/document of each word. Hence, there are more steps (some of which could be merged).

It starts with imports, then declares a singleton object (a first-class concept in Scala), with a main routine (as in Java).

The methods are colored yellow again. Note this time how dense with meaning they are this time.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

   val sc = new SparkContext(
    "local", "Inverted Index")

   sc.textFile("data/crawl")
    .map { line =>
      val array = line.split("\t", 2)
      (array(0), array(1))
    }
    .flatMap {
      case (path, text) =>
        text.split("""\W+""").map {
```


This implementation is more sophisticated than the Scalding example. It also computes the count/document of each word. Hence, there are more steps (some of which could be merged).

It starts with imports, then declares a singleton object (a first-class concept in Scala), with a main routine (as in Java).

The methods are colored yellow again. Note this time how dense with meaning they are this time.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

  val sc = new SparkContext(
   "local", "Inverted Index")

  sc.textFile("data/crawl")
   .map { line =>
    val array = line.split("\t", 2)
    (array(0), array(1))
   }
   .flatMap {
    case (path, text) =>
     text.split("""\W+""").map {
```


You being the workflow by declaring a SparkContext (in “local” mode, in this case). The rest of the program is a sequence of function calls, analogous to “pipes” we connect together to perform the data flow.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

  val sc = new SparkContext(
   "local", "Inverted Index")

  sc.textFile("data/crawl")
   .map { line =>
    val array = line.split("\t", 2)
    (array(0), array(1))
   }
   .flatMap {
    case (path, text) =>
     text.split("""\W+""").map {
```


Next we read one or more text files. If “data/crawl” has 1 or more Hadoop-style “part-NNNNN” files, Spark will process all of them (in parallel if running a distributed configuration; they will be processed synchronously in local mode).

```
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
    text.split("""\W+""") map {
      word => (word, path)
    }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
```


Now we begin a sequence of transformations on the input data.

First, we map over each line, a string, to extract the original document id (i.e., file name, UUID), 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 “String fileName, String text”.

flatMap maps over each of these 2-element tuples. We split the text into words on non-alphanumeric characters, then output collections of word (our ultimate, final “key”) and the path. Each line is converted to a collection of (word,path) pairs, so flatMap converts the collection of collections into one long “flat” collection of (word,path) pairs.

```
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
    text.split("""\W+""") map {
      word => (word, path)
    }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
```


Next, flatMap maps over each of these 2-element tuples. We split the text into words on non-alphanumeric characters, then output collections of word (our ultimate, final “key”) and the path. Each line is converted to a collection of (word,path) pairs, so flatMap converts the collection of collections into one long “flat” collection of (word,path) pairs.

```
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
    text.split("""\W+""") map {
      word => (word, path)
    }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
```


Then we map over these pairs and add a single “seed” count of 1.

```
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
.map {
  case (w, seq) =>
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }
```


reduceByKey does an implicit “group by” to bring together all occurrences of the same (word, path) and then sums up their counts.

```
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
.map {
  case (w, seq) =>
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }
```

(word1, (path1, n1))
(word2, (path2, n2))
...

Note 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)).

```
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
.map {
  c
```

(word, Seq((word, (path1, n1)), (word, (path2, n2)), ...))
...

```
    case (_, (p, n)) => (p, n)
  }
```

Thursday, November 13, 14

Now we do an explicit group by to bring all the same words together. The output will be (word, Seq((word, (path1, n1)), (word, (path2, n2)), ...)), where Seq is a Scala abstraction for sequences, e.g., Lists, Vectors, etc.

```
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
.map {
  case (w, seq) =>
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }
```

Powerful combinators

Stop for a second and admire the simplicity and elegance of this code, even if you don't understand the details. This is what coding should be, IMHO, very concise, to the point, elegant to read. Hence, a highly-productive way to work!!

```
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
.map {
  case (w, seq) =>
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }
```

Beautiful, no?

Stop for a second and admire the simplicity and elegance of this code, even if you don't understand the details. This is what coding should be, IMHO, very concise, to the point, elegant to read. Hence, a highly-productive way to work!!

$$\nabla \cdot \mathbf{D} = \rho$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

Another example of a beautiful and profound DSL, in this case from the world of my first profession, Physics: Maxwell's equations that unified Electricity and Magnetism: http://upload.wikimedia.org/wikipedia/commons/c/c4/Maxwell'sEquations.svg

```
.groupBy {
  case (w, (p, n)) => w
}
.map {
  case (w, seq) =>
    val seq2 = seq map {
      case (_, (p, n)) => (p, n)
    }.sortBy {
      case (path, n) => (-n, path)
    }
    (w, seq2.mkString(", "))
}
.saveAsTextFile(args.outpath)

sc.stop()
}
}
```

(word, “(word, (path4, n4)), (word, (path1, n1)), …”)
…

Back to the code, the last map step looks complicated, but all it does is sort the sequence by the count descending (because you would want the first elements in the list to be the documents that mention the word most frequently), then it converts the sequence into a string.

```
    .groupBy {
      case (w, (p, n)) => w
    }
    .map {
      case (w, seq) =>
        val seq2 = seq map {
          case (_, (p, n)) => (p, n)
        }.sortBy {
          case (path, n) => (-n, path)
        }
        (w, seq2.mkString(", "))
    }
    .saveAsTextFile(argz.outpath)

    sc.stop()
  }
}
```


We finish by saving the output as text file(s) and stopping the workflow.

Altogether

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(args: Array[String]) = {

  val sc = new SparkContext(
   "local", "Inverted Index")

  sc.textFile("data/crawl")
  .map { line =>
    val array = line.split("\t", 2)
    (array(0), array(1))
  }
  .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))
  }
  .groupBy {
    case (w, (p, n)) => w
  }
  .map {
    case (w, seq) =>
      val seq2 = seq map {
        case (_, (p, n)) => (p, n)
      }.sortBy {
        case (path, n) => (-n, path)
      }
      (w, seq2.mkString(", "))
  }
  .saveAsTextFile(argz.outpath)

  sc.stop()
 }
}
```


The whole shebang (12pt. font, this time)

Spark also has a streaming mode for "semi-real time" event handling.

Spark Streaming operates on data "slices" of granularity as small as 0.5-1 second. Not quite the same as single event handling, but possibly all that's needed for ~90% (?) of scenarios.

Scala for Mathematics

Spire and Algebird. ScalaZ also has some of these data structures and algorithms.

Algebird

Large-scale Analytics

https://github.com/twitter/algebird

- Algebraic types like **Monoids**:
  - A set of elements.
  - An associative binary operation (think "addition").
  - An identity element.

For big data, you're often willing to trade 100% accuracy for much faster approximations. Algebird implements many well-known approx. algorithms, all of which can be modeled generically using Monoids or similar data structures.

- Efficient approximation algorithms.
  - "Add All the Things", infoq.com/presentations/abstract-algebra-analytics

For big data, you're often willing to trade 100% accuracy for much faster approximations. Algebird implements many well-known approx. algorithms, all of which can be modeled generically using Monoids or similar data structures.

Spire

Fast Numerics

https://github.com/non/spire

- Types: Complex, Quaternion, Rational, Real, Interval, ...
- Algebraic types: Semigroups, Monoids, Groups, Rings, Fields, Vector Spaces, ...
- Trigonometric Functions.
- ...

It exploits macros with generics to implement efficient computation without requiring custom types for particular primitives.

Thursday, November 13, 14

Let's recap why is Scala taking over the big data world.

Elegant DSLs

```
...
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  case (n1, n2) => n1 + n2
}
.map {
  case ((w, p), n) => (w, (p, n))
}
.groupBy {
  case (w, (p, n)) => w
}
...
```


You would be hard pressed to find a more concise DSL for big data!!

The JVM

You have the rich Java ecosystem at your fingertips.

Functional Combinators

SQL Analogs

```
CREATE TABLE inverted_index (
  word    CHARACTER(64),
  id1     INTEGER,
  count1  INTEGER,
  id2     INTEGER,
  count2  INTEGER);
```

```
val inverted_index:
Stream[(String,Int,Int,Int,Int)]
```


You have functional “combinators”, side-effect free functions that combine/compose together to create complex algorithms with minimal effort.
For simplicity, assume we only keep the two documents where the word appears most frequently, along with the counts in each doc.
We’ll model the same data set in Scala with a Stream, because we’re going to process it in “batch”.

Functional Combinators

```sql
SELECT * FROM inverted_index
WHERE word = 'Scala';
```

Restrict

```scala
inverted_index.filter {
  case (word, count) =>
    word == "Scala"
}
```


The equivalent of a select statement in both SQL and Scala.

Functional Combinators

```
SELECT word FROM inverted_index;
```

Projection

```
inverted_index.map {
  case (word, _,_,_,_) =>
    word
}
```


Projecting just the fields we want.

Functional Combinators

```
SELECT count, size(word) AS size
FROM inverted_index
GROUP BY count1
ORDER BY size DESC;
```

Group By and Order By

```
inverted_index.groupBy {
  case (word, _, count, _, _) => count
}.toList.map {
  case (count, words) => (count, words.size)
}.sortBy {
 case (count, size) => -size
}
```


Group By: group by the frequency of occurrence for the first document, then order by the group size, descending.

What could be improved?

- More hardening of Spark and alternatives.
- Need an "iPython" for Scala.
  - github.com/Bridgewater/scala-notebook
- Need data visualization tools.

Spark is already good, but it hasn't been battle tested to the same extent as MR, especially at very large scales, e.g., Petabyte number crunching.
Data Scientists and many "data-centric" developers like us need interactive tools with not only code, but also data graphs and other visualization tools.
We also need more sophisticated viz tools integrated with Scala libraries. There are some open-source projects working on this need.

- How to enforce Schema at scale?
  - See what Spark SQL is doing.
- Better primitive operation support for data crunching.
- Better JVM optimizations.
- ... and other JVM languages should join us!

Is it too expensive to create a tuple or case class for each record? It's fine to use these convenient constructs to define record schemas, but at scale, we want to work with arrays of primitives for optimal performance.
Speaking of which, we could use unsigned types and we would like to see the problem of collection specialization solved for good (mini-boxing?), if possible.
At the JVM level, value types (on stack rather than on heap) and unsigned primitives would help.

Finally, Clojure, Java 8, etc. should be prominent here, too!

dean.wampler@typesafe.com
polyglotprogramming.com/talks
@deanwampler
Typesafe
SOME RIGHTS RESERVED