NE Scala Symposium
@deanwampler
March 9, 2012

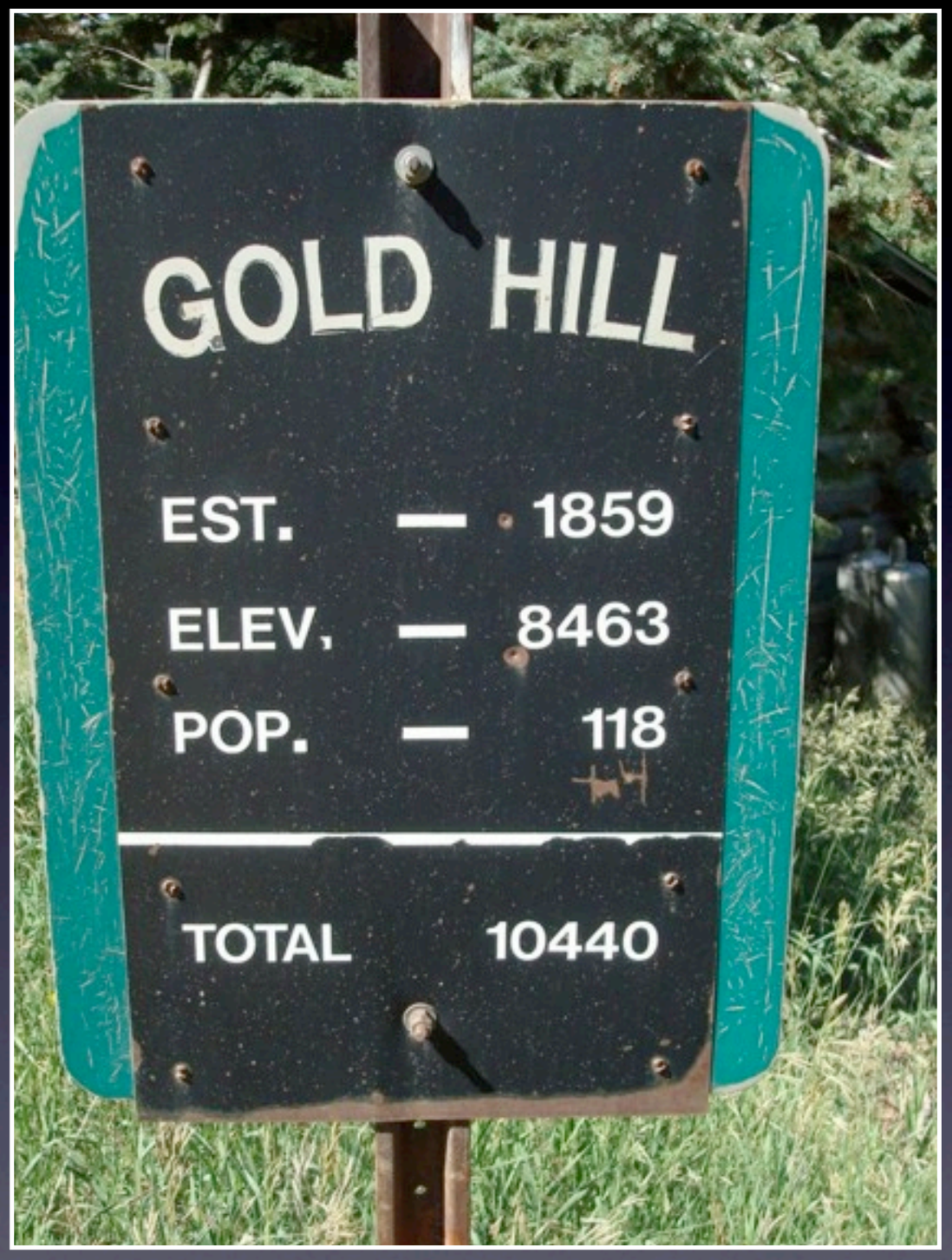
Why Big Data Needs to Be Functional

SOME RIGHTS RESERVED

All pictures © Dean Wampler, 2011–2012.

What is Big Data?

Data so big that traditional solutions are too slow, too small, or too expensive to use.



Hat tip: Bob Korbus

It's a buzz word, but generally associated with the problem of data sets too big to manage with traditional SQL databases. A parallel development has been the NoSQL movement that is good at handling semistructured data, scaling, etc.

Friday, March 16, 12

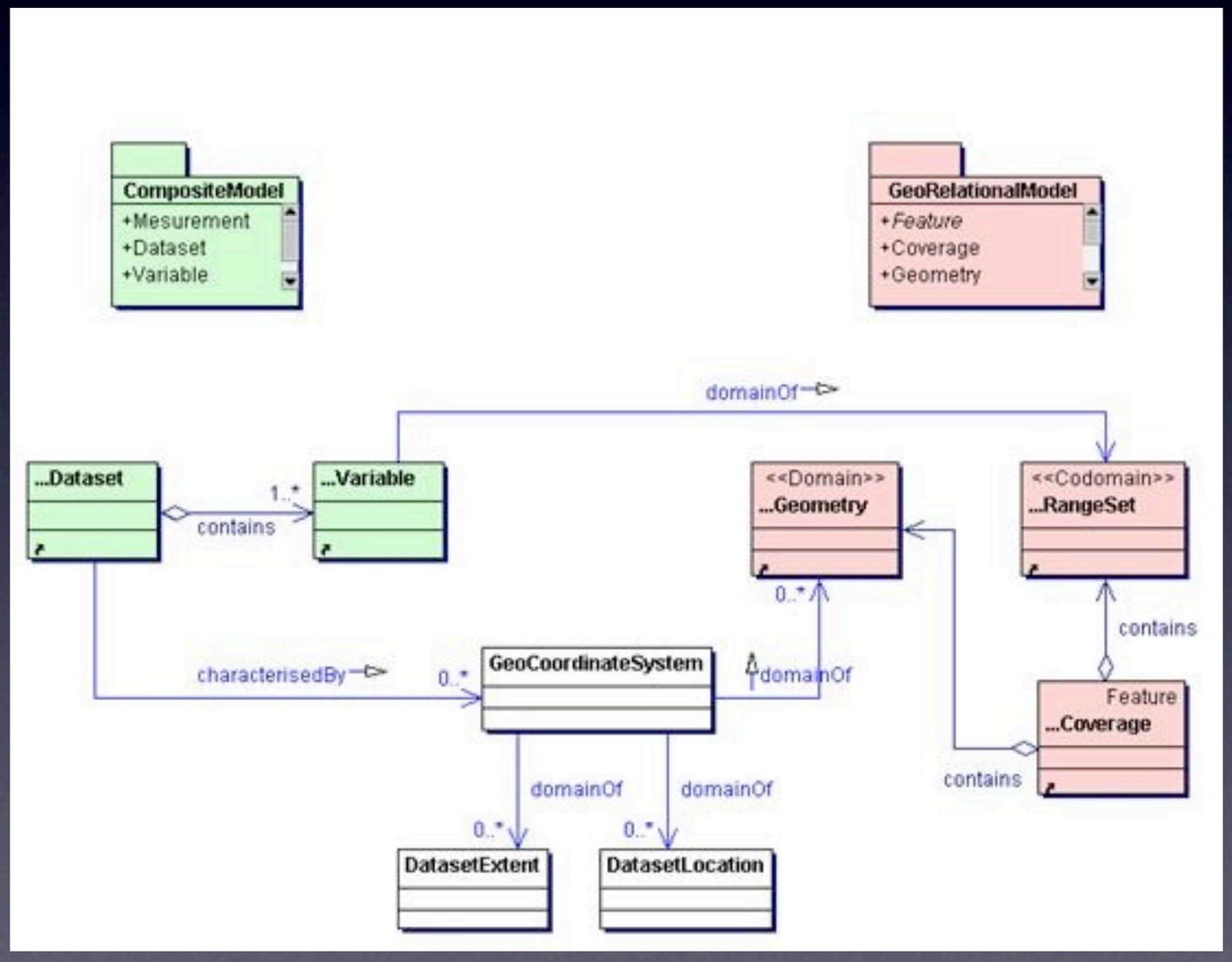
Three trends influence my thinking...

Data Size ⬆

Friday, March 16, 12

Data volumes are obviously growing… rapidly.

Formal Schemas ⬇



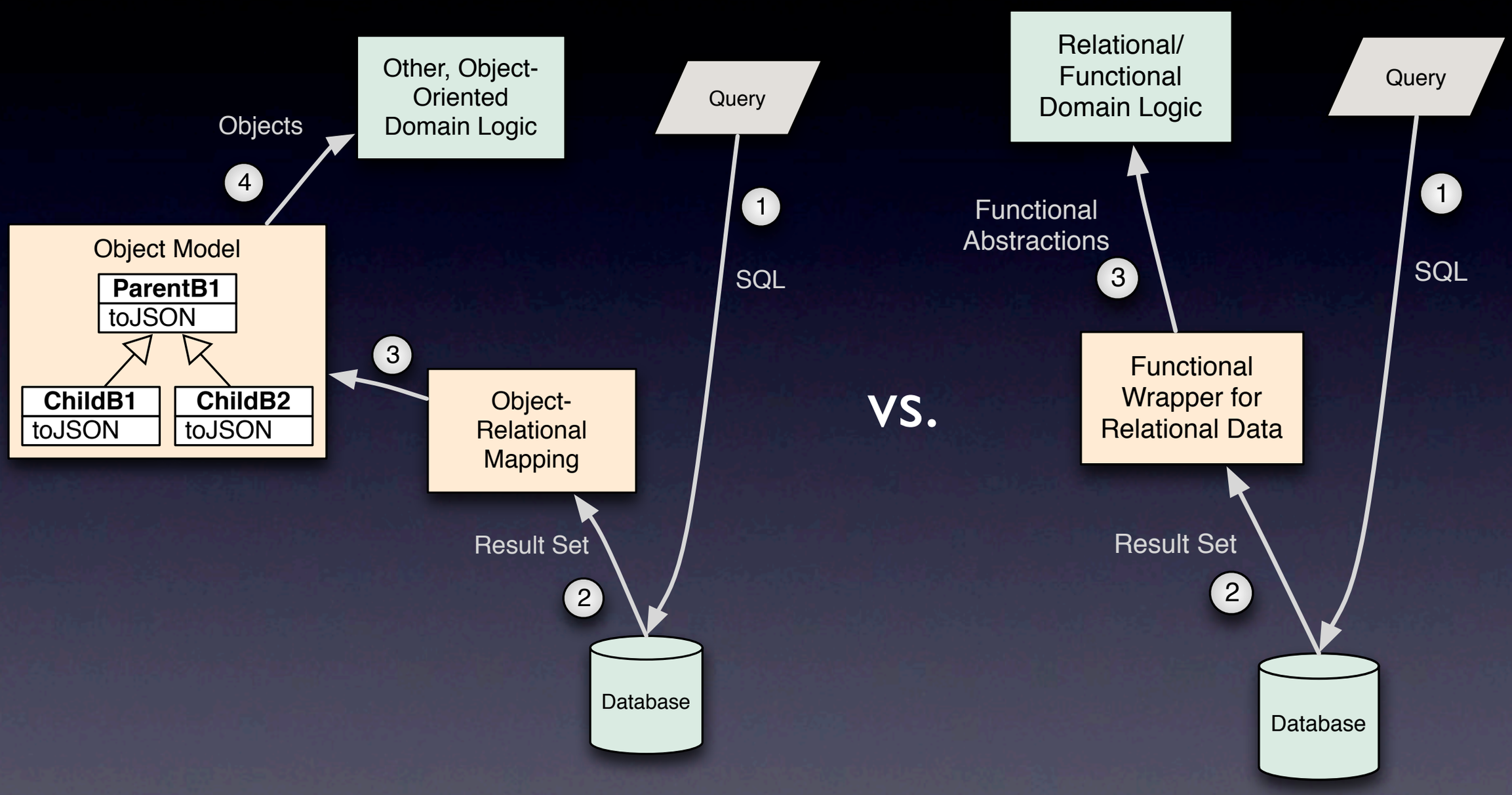
Friday, March 16, 12

There is less emphasis on “formal” schemas and domain models, because data changes rapidly, there are disparate data sources being joined, using relatively-agnostic software (e.g., collections of things where the software is agnostic about the contents) tends to be faster to develop and run.

Data-Driven Programs ⬆

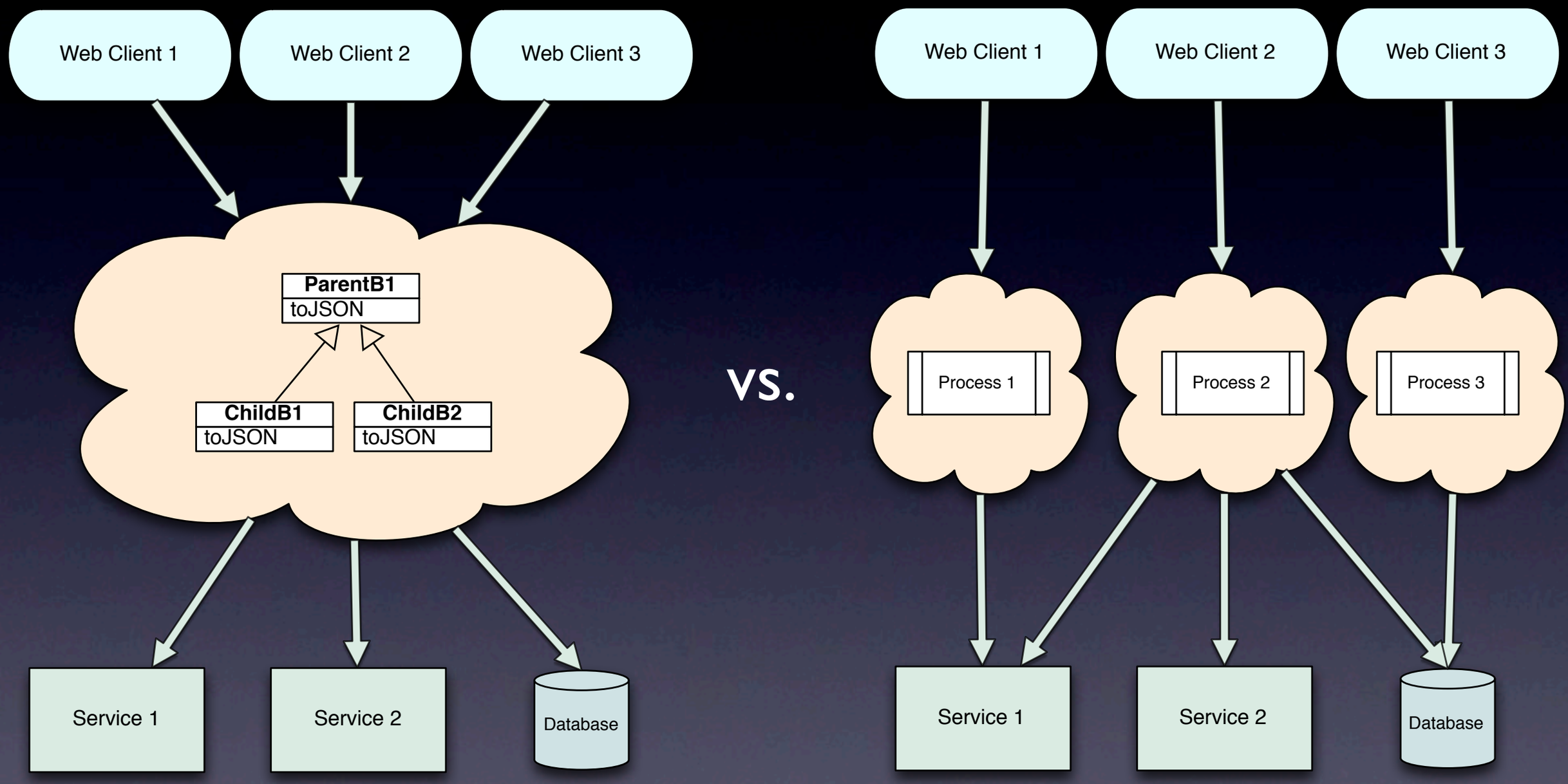
Friday, March 16, 12

Machine learning is growing in importance. Here, generic algorithms and data structures are trained to represent the “world” using data, rather than encoding a model of the world in the software itself.



Friday, March 16, 12

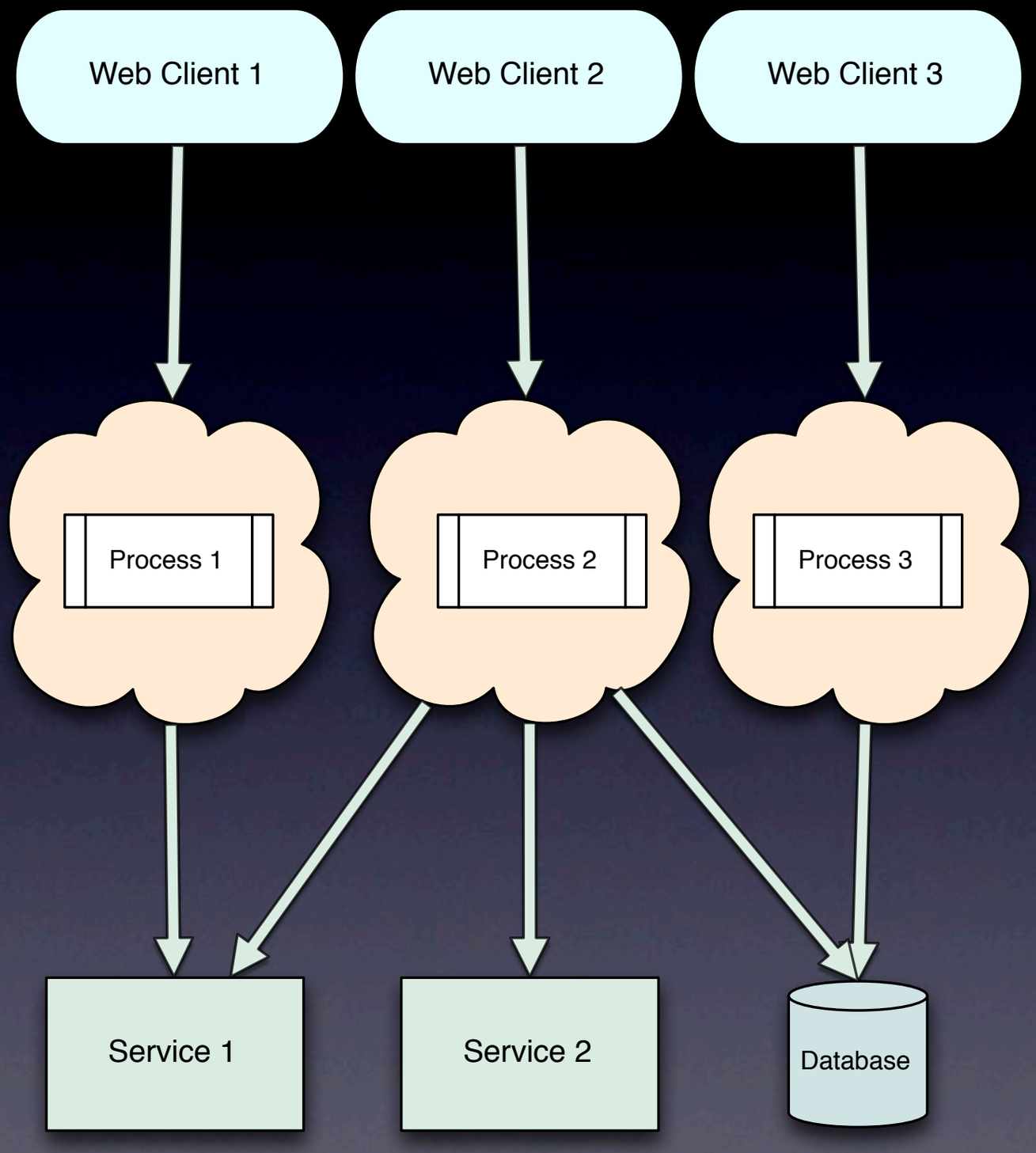
Traditionally, (on the left) we've kept a rich, in-memory domain model requiring an ORM to convert persistent data into the model. This is resource overhead and complexity we can't afford in big data systems. Rather, (on the right) we should treat the result set as it is, a particular kind of collection, do the minimal transformation required to exploit our collections libraries and classes representing some domain concepts (e.g., Address, StockOption, etc.), then write functional code to implement business logic (or drive emergent behavior with machine learning algorithms…)



Friday, March 16, 12

In a broader view, object models (on the left) tend to push us towards centralized, complex systems that don't decompose well and stifle reuse and optimal deployment scenarios. FP code makes it easier to write smaller, focused services (on the right) that we compose and deploy as appropriate.

- Data Size ⬆
- Formal Schema ⬇
- Data-Driven Programs ⬆



Friday, March 16, 12

Smaller, focused services scale better, especially horitzontally. They also don’t encapsulate more business logic than is required, and this (informal) architecture is also suitable for scaling ML and related algorithms.

Using Scala for MapReduce

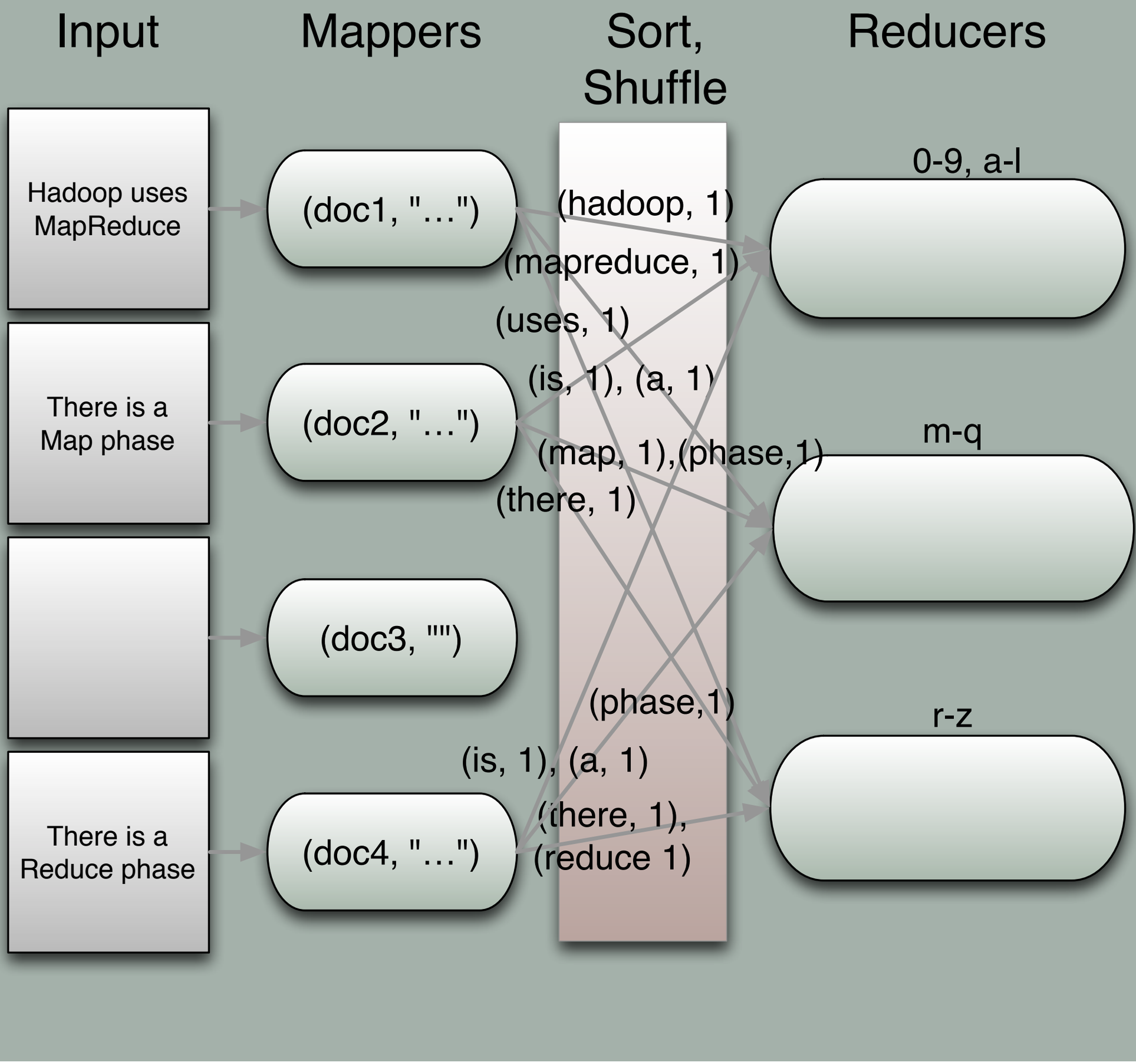
Friday, March 16, 12

Back to Scala, let's look at the mess that Java has introduced into Hadoop MapReduce, and get a glimpse of a better way.
As an example, I'll walk you through the "Hello World" of MapReduce; the Word Count algorithm...

Consider *Word Count*

Each document gets a mapper. I'm showing the document contents in the boxes for this example. Actually, large documents might get split to several mappers (as we'll see). It is also possible to concatenate many small documents into a single, larger document for input to a mapper.

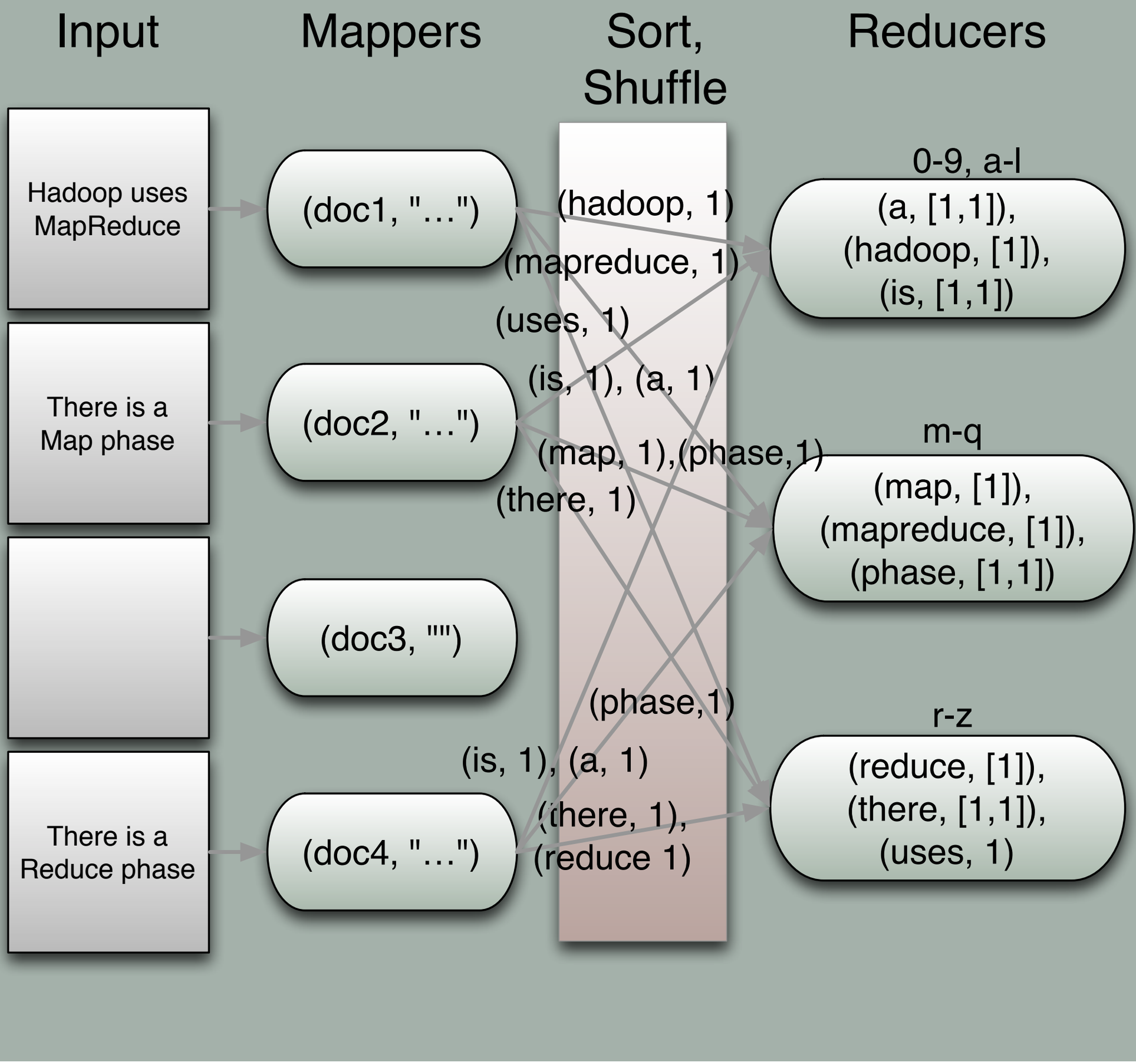
Each mapper will receive a key-value pair, where the key is the document path and the value is the contents of the document. It will ignore the key, tokenize the content, convert all words to lower case and count them...



The mappers emit key-value pairs, where each key is one of the words, and the value is the count. In the most naive (but also most memory efficient) implementation, each mapper simply emits (word, 1) each time “word” is seen.

The mappers themselves don’t decide to which reducer each pair should be sent. Rather, the job setup configures what to do and the Hadoop runtime enforces it during the Sort/Shuffle phase, where the key-value pairs in each mapper are sorted by key (that is locally, not globally or “totally”) and then the pairs are routed to the correct reducer, on the current machine or other machines.

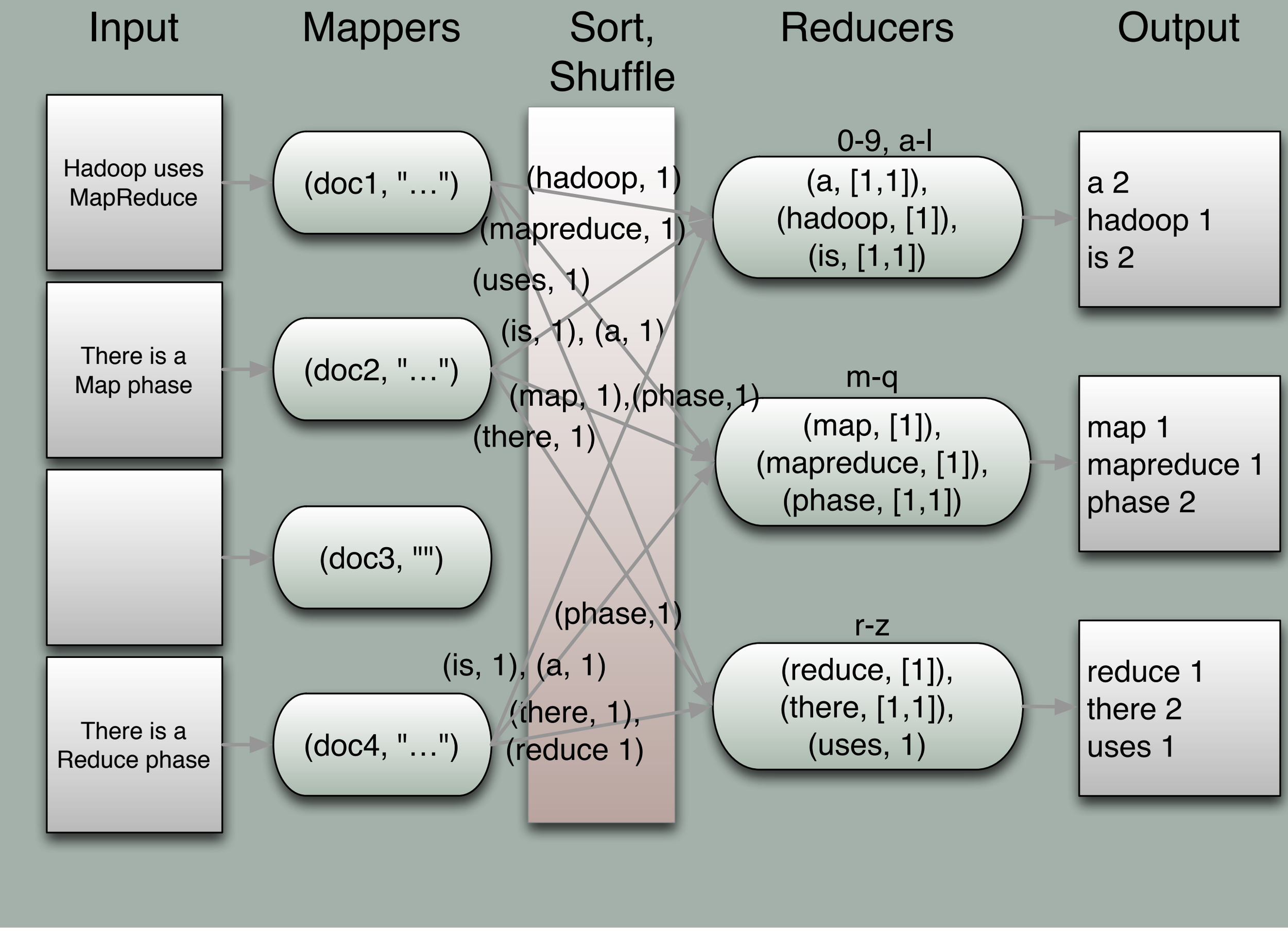
Note how we partitioned the reducers (by first letter of the keys). Also, note that the mapper for the empty doc. emits no pairs, as you would expect.



The mappers emit key-value pairs, where each key is one of the words, and the value is the count. In the most naive (but also most memory efficient) implementation, each mapper simply emits (word, 1) each time “word” is seen.

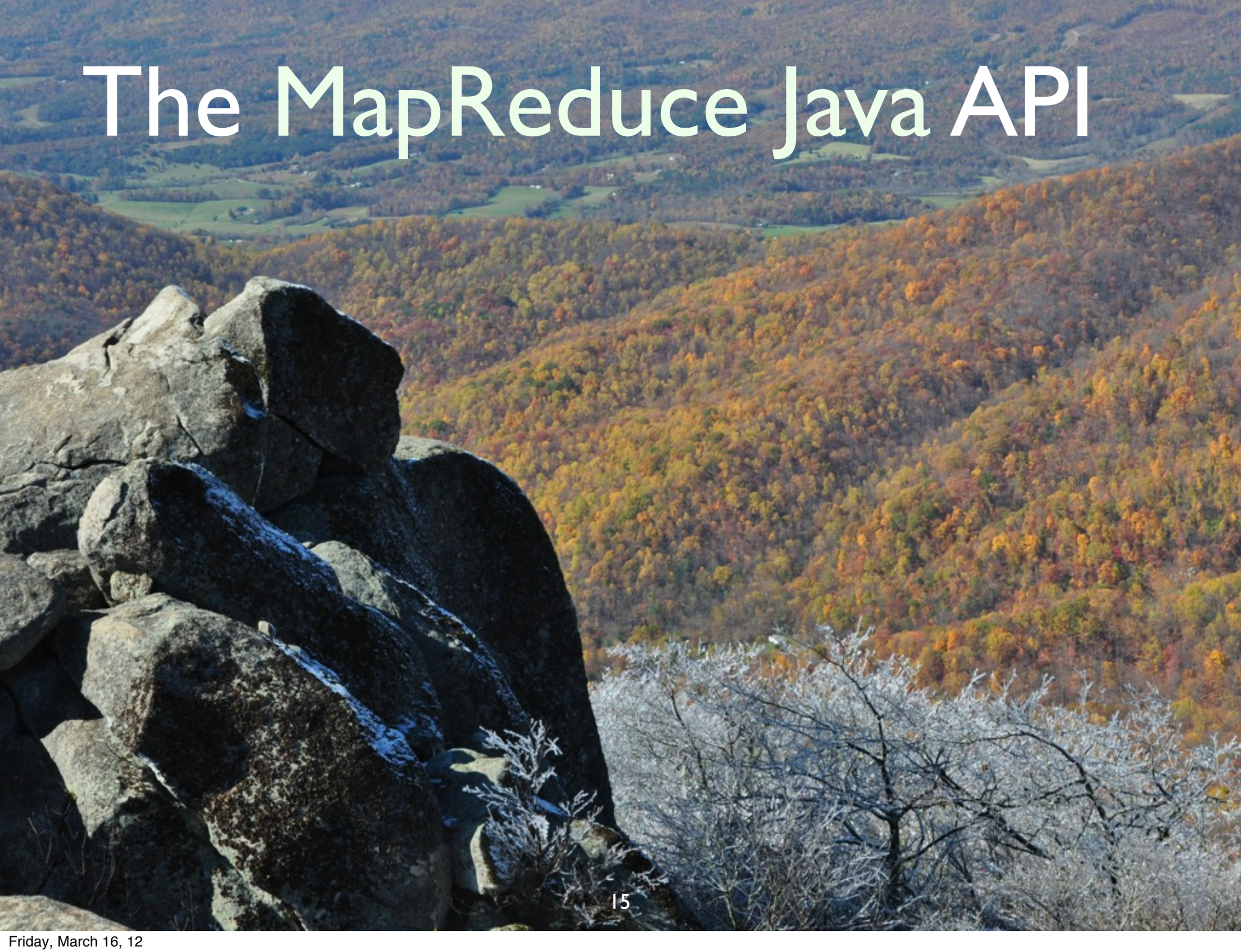
The mappers themselves don’t decide to which reducer each pair should be sent. Rather, the job setup configures what to do and the Hadoop runtime enforces it during the Sort/Shuffle phase, where the key-value pairs in each mapper are sorted by key (that is locally, not globally or “totally”) and then the pairs are routed to the correct reducer, on the current machine or other machines.

Note how we partitioned the reducers (by first letter of the keys). Also, note that the mapper for the empty doc. emits no pairs, as you would expect.



The final view of the WordCount process flow for our example.
We'll see in more detail shortly how the key-value pairs are passed to the reducers, which add up the counts for each word (key) and then writes the results to the output files.
The the output files contain one line for each key (the word) and value (the count), assuming we're using text output. The choice of delimiter between key and value is up to you. (We'll discuss options as we go.)

The MapReduce Java API



A Java API, but I'll show you Scala code; see my GitHub scala-hadoop project.

```
import org.apache.hadoop.io._
import org.apache.hadoop.mapred._
import java.util.StringTokenizer

object SimpleWordCount {

  val one  = new IntWritable(1)
  val word = new Text     // Value will be set in a non-thread-safe way!

  class WCMapper extends MapReduceBase with Mapper[LongWritable, Text, Text, IntWritable] {

    def map(key: LongWritable, valueDocContents: Text,
            output: OutputCollector[Text, IntWritable], reporter: Reporter): Unit = {
      val tokens = valueDocContents.toString.split("\\s+")
      for (wordString <- tokens) {
        if (wordString.length > 0) {
          word.set(wordString.toLowerCase)
          output.collect(word, one)
        }
      }
    }

  class Reduce extends MapReduceBase with Reducer[Text, IntWritable, Text, IntWritable] {

    def reduce(keyWord: Text, valuesCounts: java.util.Iterator[IntWritable],
               output: OutputCollector[Text, IntWritable], reporter: Reporter): Unit = {
      var totalCount = 0
      while (valuesCounts.hasNext) {
          totalCount += valuesCounts.next.get
       }
       output.collect(keyWord, new IntWritable(totalCount))
    }
  }
}
```


Friday, March 16, 12

This is intentionally too small to read. The algorithm is simple, but the framework is in your face. Note all the green types floating around and relatively few yellow methods implementing actual operations. Still, I've omitted many boilerplate details for configuring and running the job. This is just the "core" MapReduce code. In fact, Word Count is not too bad, but when you get to more complex algorithms, even conceptually simple things like relational-style joins, code in this API gets complex and tedious very fast.

Using Crunch (Java)

Crunch is a Java library that provides a higher-level abstraction for data computations and flows on top of MapReduce, inspired by a paper from Google researchers on an internal project called FlumeJava.
See https://github.com/cloudera/crunch.

Crunch Concepts

- *Pipeline<T>*: Abstraction for data processing flow.
- *PCollection<T>*: Distributed, unordered collection of elements T.
- *parallelDo()*: Apply operation across collection.

Friday, March 16, 12

A quick overview of Crunch concepts. Note that it's necessary for Crunch to provide features missing in Java: parallel data structures and operations over them. (Even non-parallel data structures are poor, missing the "power tools" of functional programming, operations like map, flatMap, fold, etc.)

Crunch Concepts

- *PTable<K, V>*: Distributed multimap.
  - *groupByKey()*: group together all values with the same key.
  - *parallelDo()*: apply operation across collection.

Crunch Concepts

- *PGroupedTable<K, V>*: Output of *groupByKey()*.
  - *combineValues()*: associative and commutative .
  - *groupByKey()*: group together all values with the same key.
- *parallelDo()*: apply operation across collection.

```
import com.cloudera.crunch.*;
import org.apache.hadoop.*;
...

public class WordCount extends Configured implements Tool, Serializable {

  public int run(String[] args) throws Exception {
    Pipeline pipeline = new MRPipeline(WordCount.class, getConf());
    PCollection<String> lines = pipeline.readTextFile(args[0]);

    PCollection<String> words = lines.parallelDo(new DoFn<String, String>() {
      public void process(String line, Emitter<String> emitter) {
        for (String word : line.split("\\s+")) {
          emitter.emit(word);
        }
      }
    }, Writables.strings());

    PTable<String, Long> counts = Aggregate.count(words);
    pipeline.writeTextFile(counts, args[1]);
    pipeline.done();
    return 0;
  }
}
```


This is back to Java (rather than Scala calling a Java API). I omitted the setup and “main” again, but that code is a lot smaller than for generic MapReduce.
It’s a definite improvement, especially for more sophisticated algorithms. Crunch (and FlumeJava), as well as similar toolkits like Cascading, are steps in the right direction, but there is still 1) lots of boilerplate, 2) functional ideas crying to escape the Java world view, and poor support in the MapReduce API (like no Tuple type) that harms Crunch, etc. Still, note that relatively heavy amount of type information compared to methods (operations). It would improve some if I wrote this code in Scala, thereby exploiting type inference, but not by a lot.

Using Scrunch (Scala)

Friday, March 16, 12

Scrunch is a scala DSL around Crunch writing by the same developers at Cloudera and also included in the Crunch distro. It uses type classes and similar constructs to provide wrap the Crunch classes and MR classes with REAL map, flatMap, etc. functionality.

```
import com.cloudera.crunch._
import com.cloudera.scrunch._
...

class ScrunchWordCount {
  def wordCount(inputFile: String,
                outputFile: String) = {

    val pipeline = new Pipeline[ScrunchWordCount]
    pipeline.read(from.textFile(inputFile))
      .flatMap(_.toLowerCase.split("\\W+"))
      .filter(!_.isEmpty())
      .count
      .write(to.textFile(outputFile)) // Word counts
      .map((w, c) => (w.slice(0, 1), c))
      .groupByKey.combine(v => v.sum).materialize
    pipeline.done
  }
}

object ScrunchWordCount {
  def main(args: Array[String]) = {
    new ScrunchWordCount.wordCount(args(0), args(1))
  }
}
```


Friday, March 16, 12

(Back to Scala) I cheated; I'm showing you the WHOLE program, "main" and all. Not only is the size significantly smaller and more concise still, but note the "builder" style notation, which intuitive lays out the data flow required. There is must less green - fewer types are explicitly tossed about, and not just because Scala does implicit typing. In contrast, there is more yellow - function calls showing the sequence of operations that more naturally represent the real "business logic", i.e., the data flow that is Word Count!
Also, fewer comments are required to help you sort out what's going on. Once you understand the meaning of the individual, high-reusable operations like flatMap and filter, you can construct arbitrarily-complex transformations and calculations with relative ease.

You Also See this Functional Improvement with...

- *Cascading* (Java) vs.
- *Cascalog* (Clojure)
- *Scalding* (Scala)

Friday, March 16, 12

Similarly, for the better-known Cascading Java API on top of Hadoop, similar “improvements” occur when you use Cascalog or Scalding.

Friday, March 16, 12

This is obvious for this crowd, but it's under-appreciated by most people in the big data world. Functional programming is ideal for data transformations, filtering, etc. Ad-hoc object models are not "the simplest thing that could possible work" (an Agile catch phrase), at least for data-oriented problems. There's a reason that SQL has been successful all these years; the relational model is very functional and fits data very well.

A Manifesto...

Friday, March 16, 12

So, I think we have an opportunity...

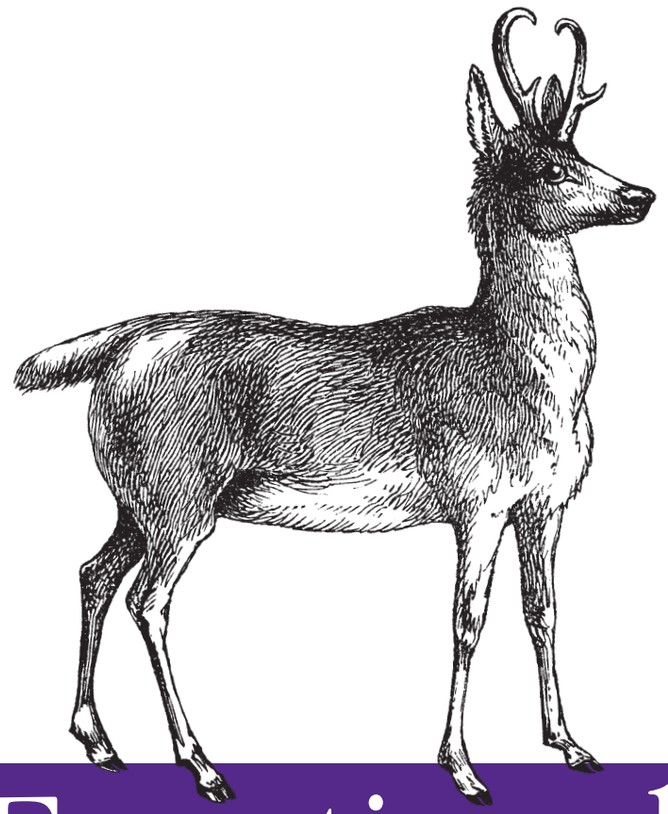
I worked with EJBs a decade ago. The framework was completely invasive into your business logic. There were too many configuration options in XML files. The framework “paradigm” was a poor fit for most problems (like soft real time systems and most algorithms beyond Word Count). Internally, EJB implementations were inefficient and hard to optimize, because they relied on poorly considered object boundaries that muddled more natural boundaries. (I’ve argued in other presentations and my “FP for Java Devs” book that OOP is a poor modularity tool…)

The fact is, Hadoop reminds me of EJBs in almost every way. It works okay and people do get stuff done, but just as the Spring Framework brought an essential rethinking to Enterprise Java, I think there is an essential rethink that needs to happen in Big Data. The Scala community is well positioned to create it.

Scala Collections.

We already have the write model in Scala's collections and the parallel versions already support multi-core horizontal scaling. With an extension to distributed horizontal scaling, they will be the ideal platform for diverse services, including those poorly served by Hadoop...

Akka is the right platform for distributed services. It exposes clean, low-level primitives for robust, distributed services (e.g., Actors), upon which we can build flexible big data systems that can handle soft real time and batch processing efficiently and scalably.
(No, this isn't Akka Mountain in Sweden. So sue me… ;)



Functional Programming

for Java Developers

O'REILLY®

Dean Wampler

FP for Java Devs

1/2 off today!!

Friday, March 16, 12

That's it. Today only (3/9), you can get my ebook 1/2 off!