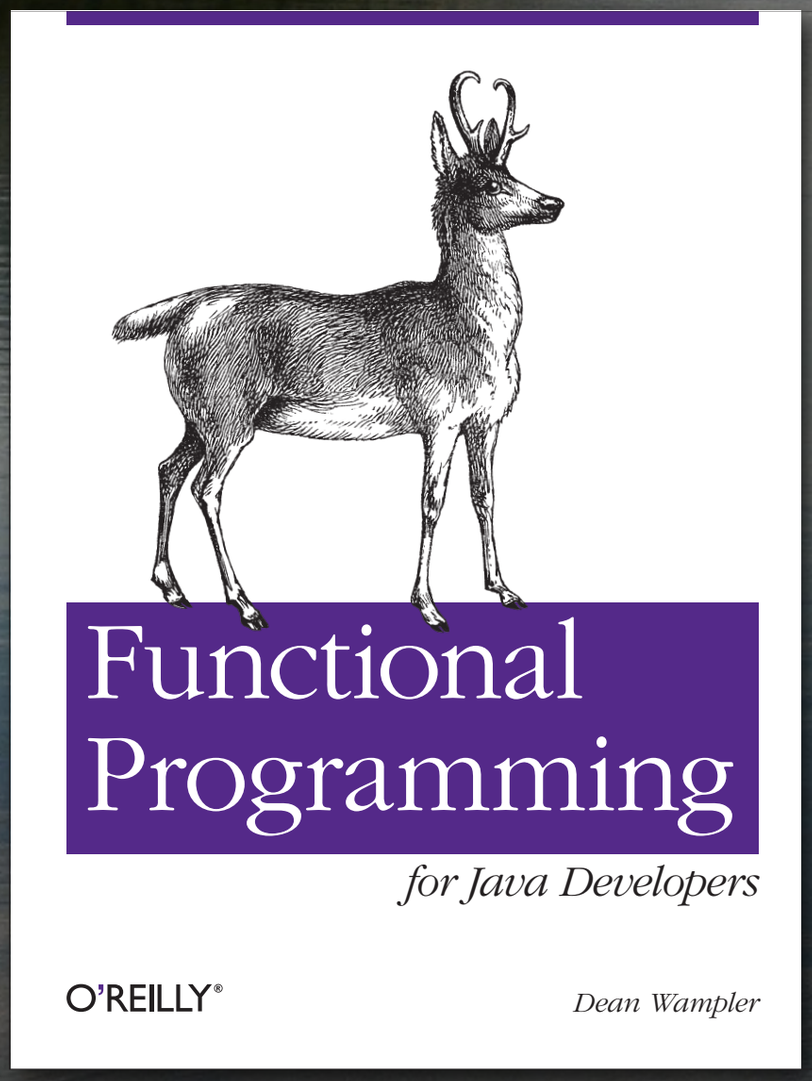
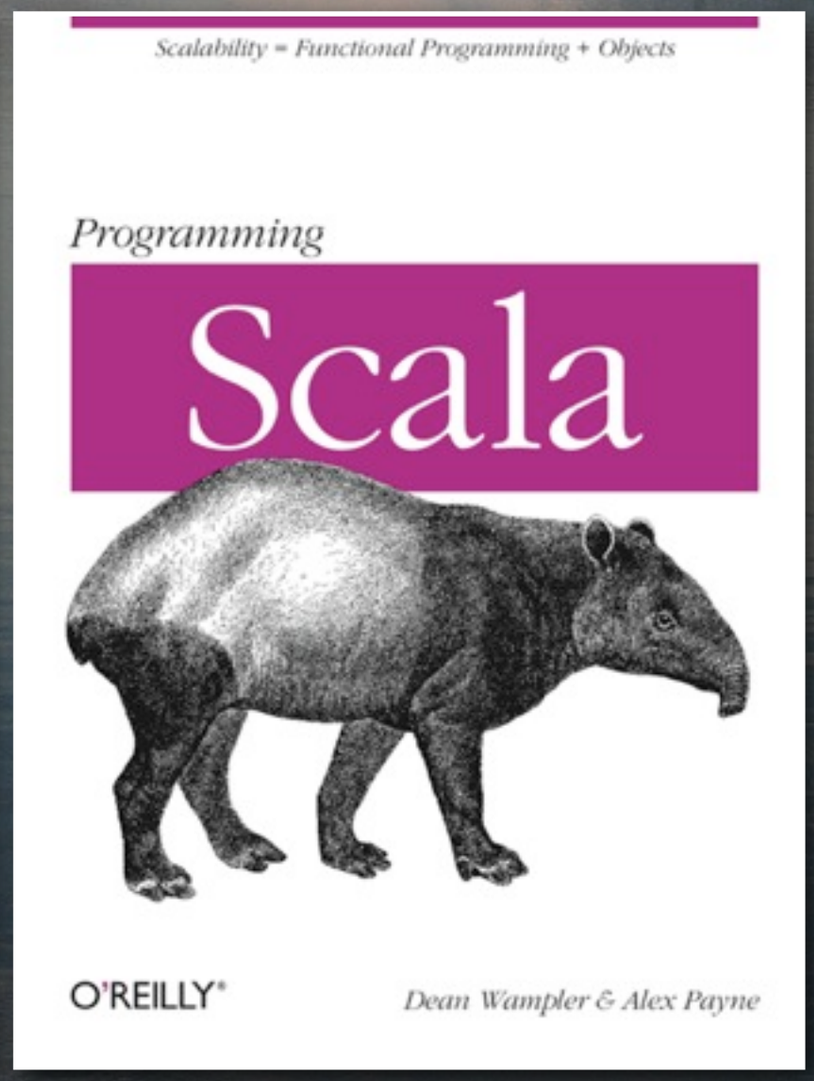
QCon NYC
June 19, 2012
@deanwampler
thinkbiganalytics.com

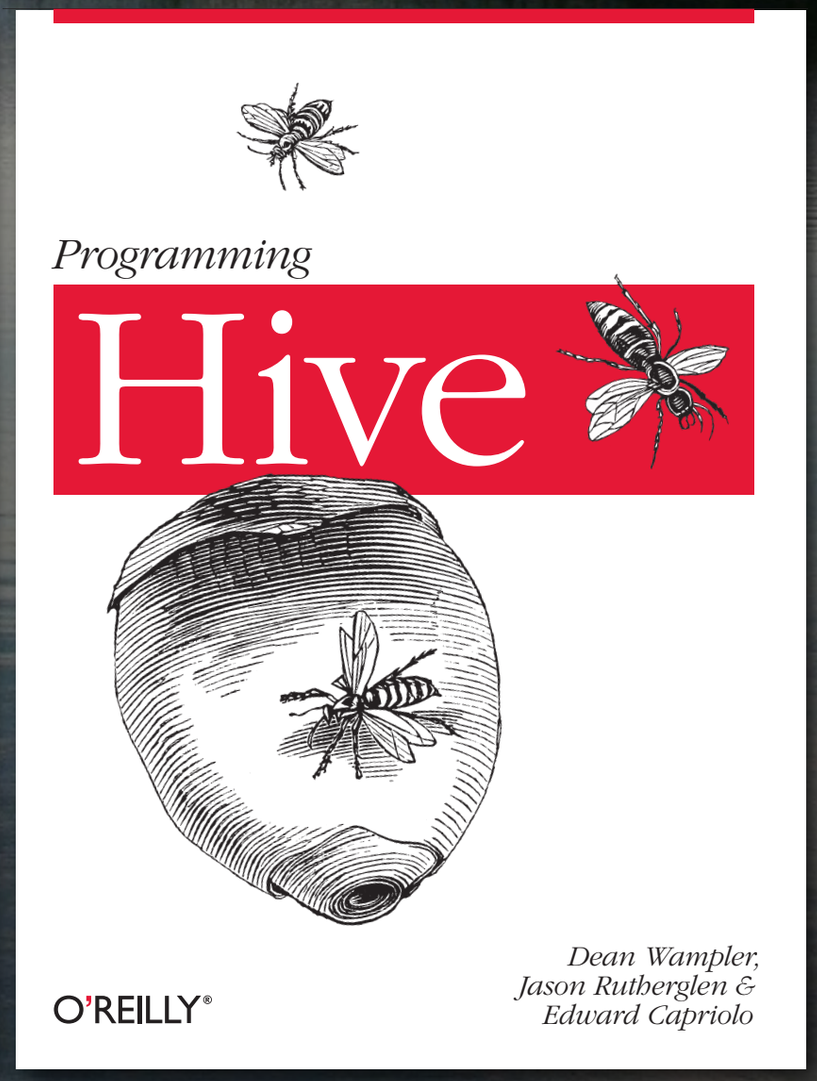
MapReduce and Its Discontents

MapReduce is a powerful tool for scaling data analytics and storage. It is on the hype curve of popularity. However, is MR the end of the story? Does it meet all our needs?

<shameless/>







"Programming Hive" is forthcoming, Sept. 2012.

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.

3 Trends

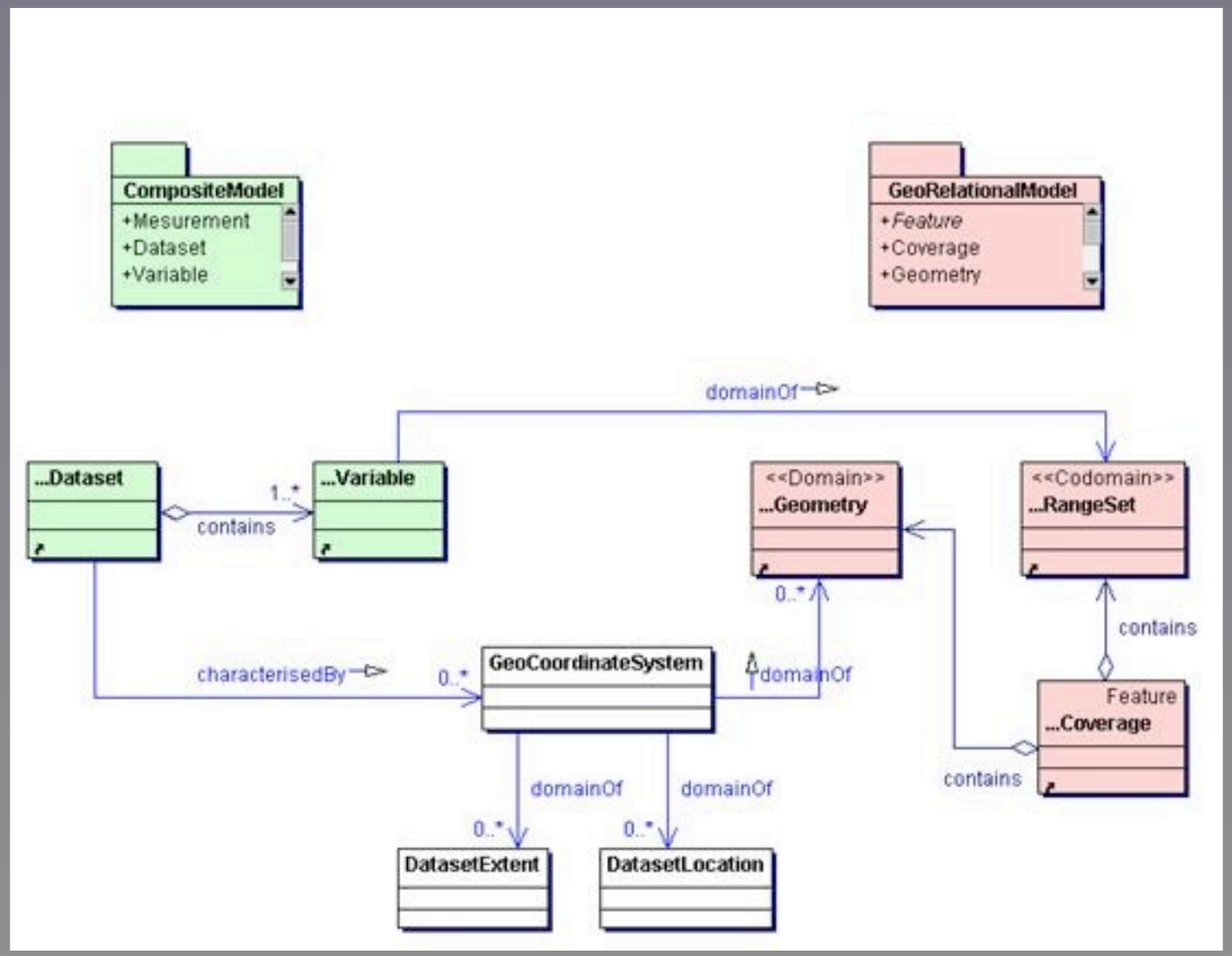
Tuesday, June 19, 12

Three trends influence my thinking...

Data Size ⬆

Data volumes are obviously growing… rapidly.

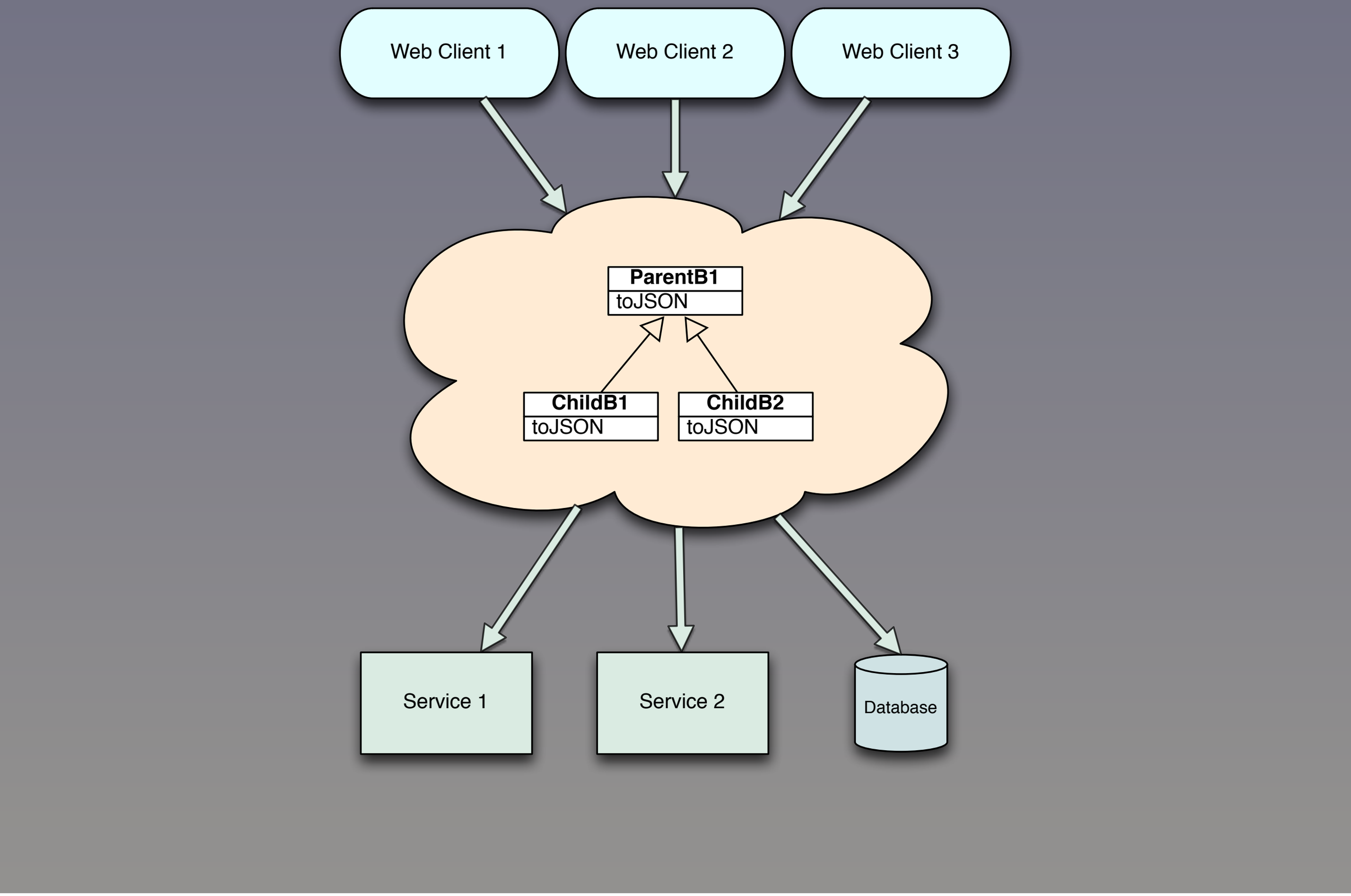
Formal Schemas ⬇



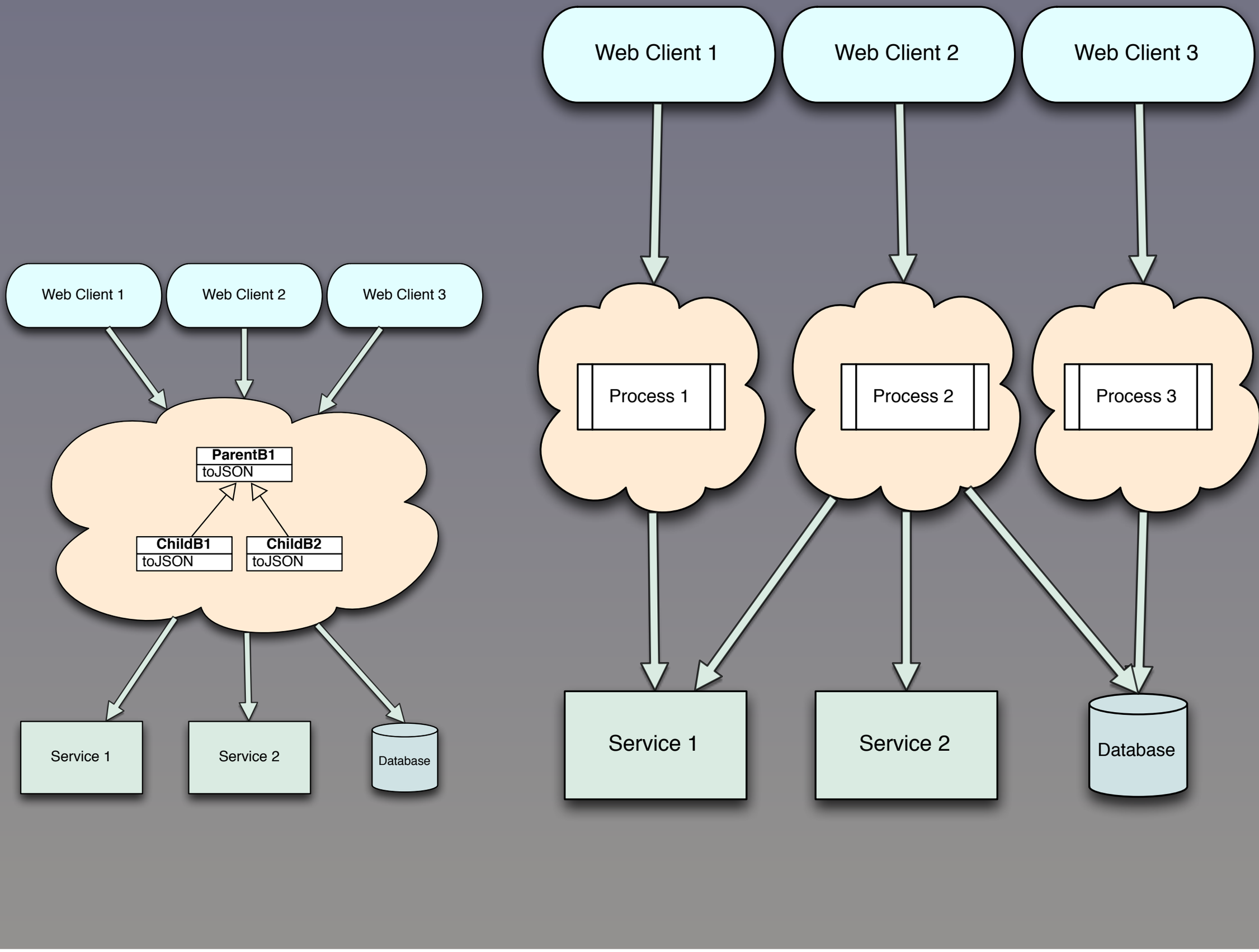
There is less emphasis on “formal” schemas and domain models, i.e., both relational models of data and OO models, because data schemas and sources change rapidly. So, using relatively-agnostic software (e.g., collections of things where the software is agnostic about the contents) tends to be faster to develop and run. Put another way, we find it more useful to build somewhat agnostic applications and drive their behavior through data.

Data-Driven Programs ⬆

This is the 2nd generation “Stanley”, the most successful self-driving car ever built (by a Google-Stanford) team. 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. It’s another example of generic algorithms that produce the desired behavior by being application agnostic and data driven, rather than hard-coding a model of the world. (In practice, however, a balance is struck between completely agnostic apps and some engineering towards for the specific problem.

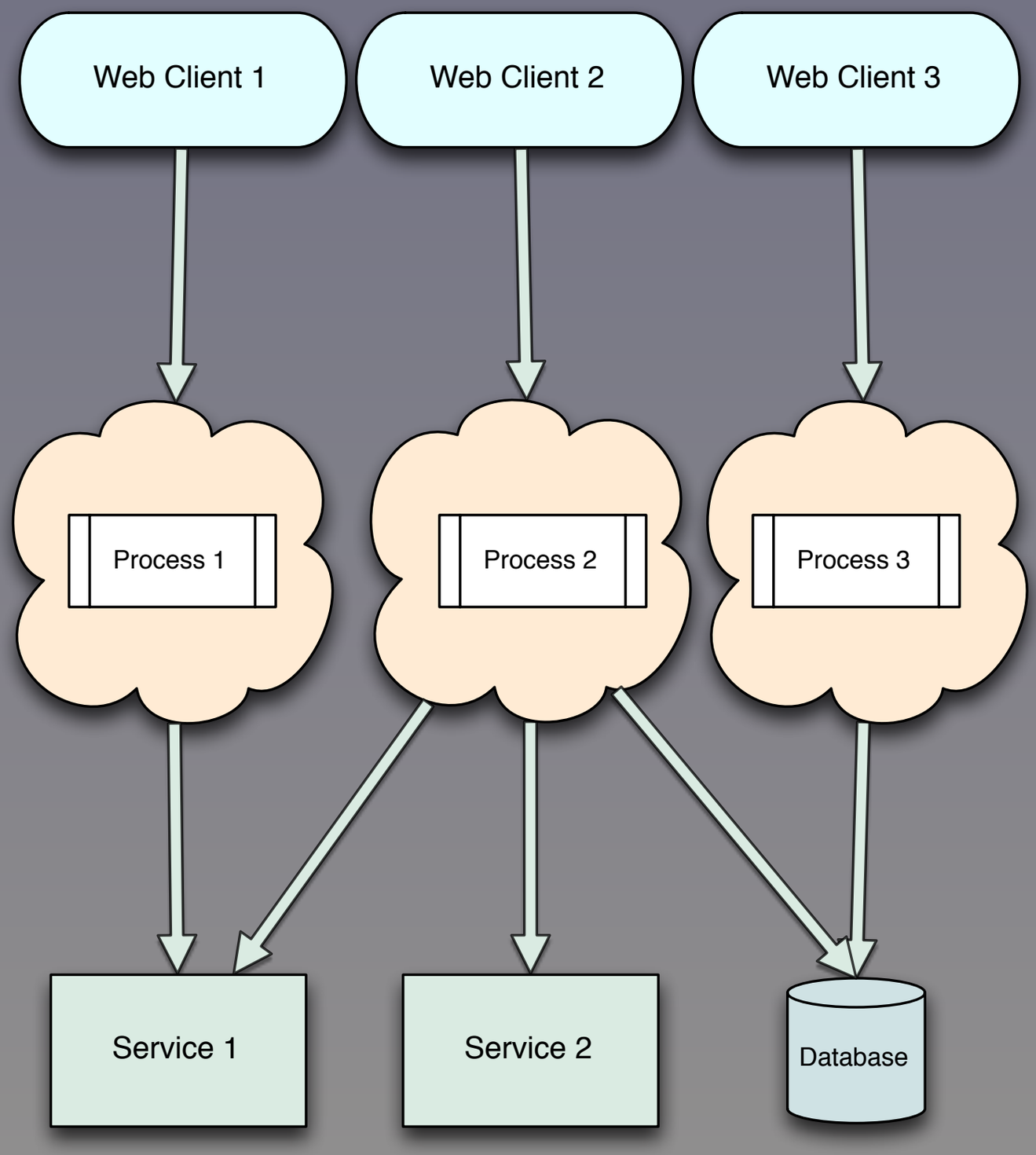


In a broader view, object models tend to push us towards centralized, complex systems that don’t decompose well and stifle reuse and optimal deployment scenarios. FP code makes it easier to write smaller, focused services that we compose and deploy as appropriate.



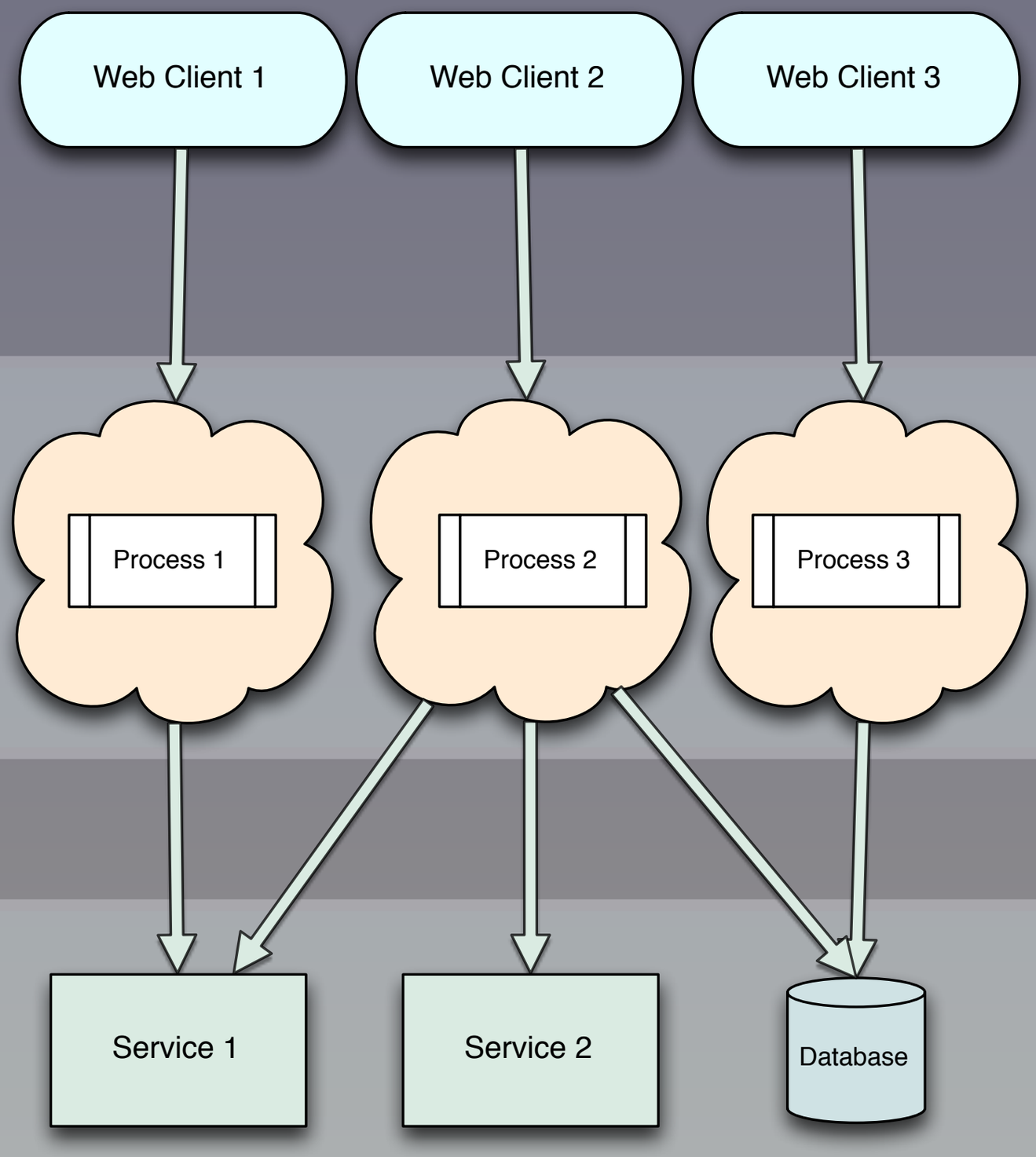
In a broader view, object models tend to push us towards centralized, complex systems that don't decompose well and stifle reuse and optimal deployment scenarios. FP code makes it easier to write smaller, focused services that we compose and deploy as appropriate.

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



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.

- MapReduce



- Distrib. FS

And MapReduce + a distributed file system, like the Hadoop DFS, fit this model.

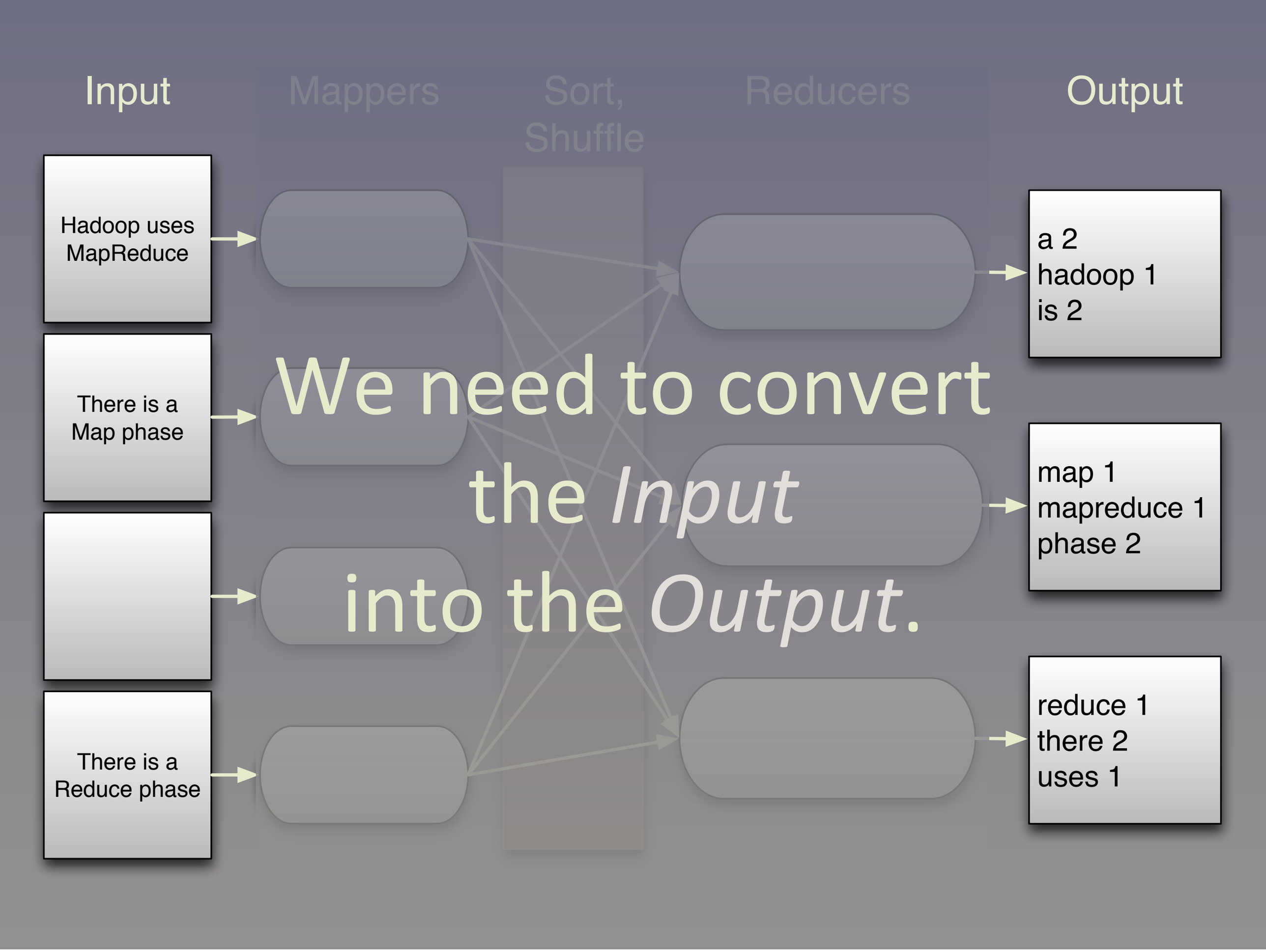
Let's walk through the "Hello World" of MapReduce; the Word Count algorithm...

MapReduce in Hadoop

Let's look at a *MapReduce* algorithm: *WordCount*.

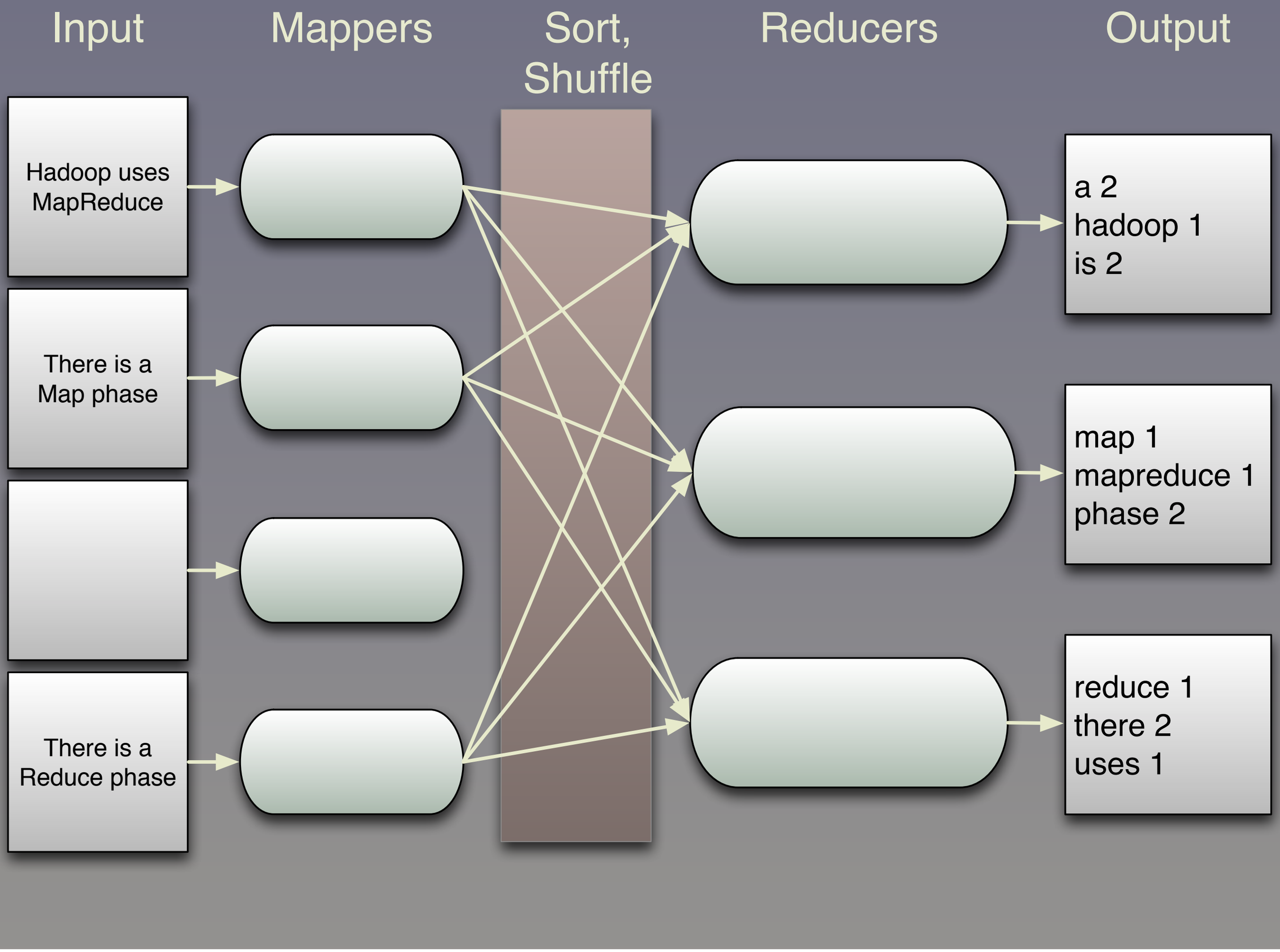
(The *Hello World* of big data...)

Let's walk through this algorithm at a conceptual level...



Tuesday, June 19, 12

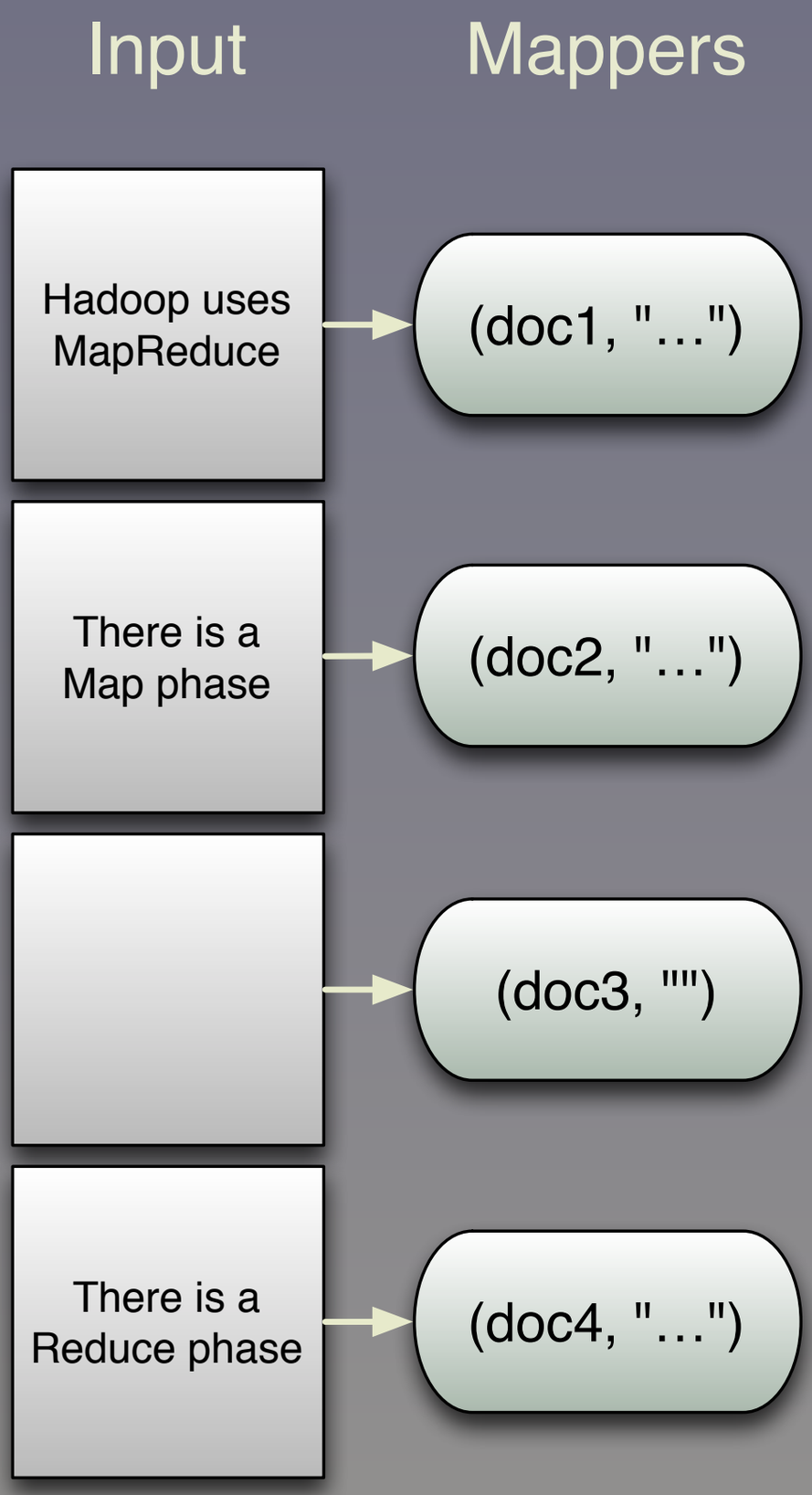
Four input documents, one left empty, the others with small phrases (for simplicity…). The word count output is on the right (we'll see why there are three output "documents"). We need to get from the input on the left-hand side to the output on the right-hand side.



Here is a schematic view of the steps in Hadoop MapReduce. Each Input file is read by a single Mapper process (default: can be many-to-many, as we'll see later).

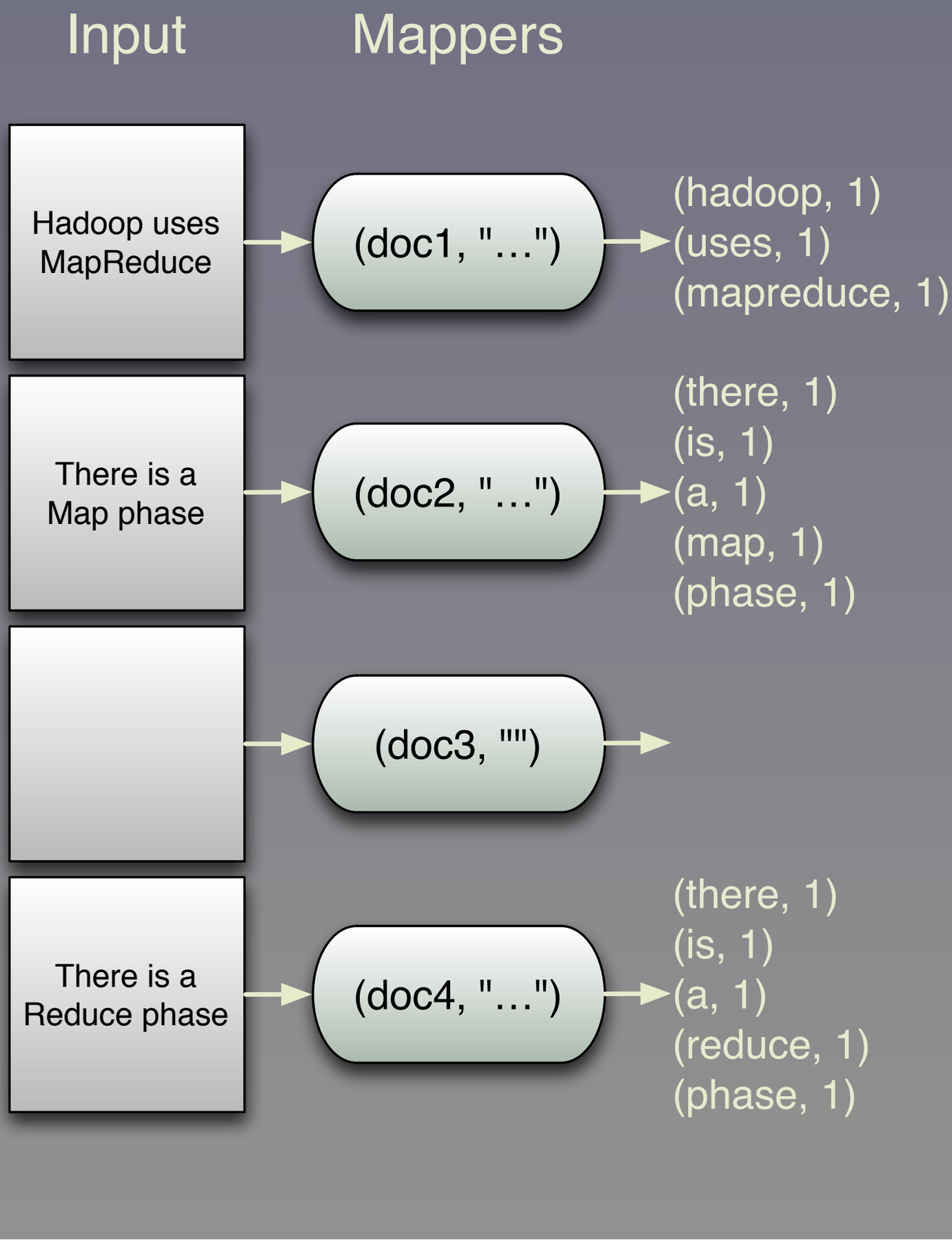
The Mappers emit key-value pairs that will be sorted, then partitioned and "shuffled" to the reducers, where each Reducer will get all instances of a given key (for 1 or more values).

Each Reducer generates the final key-value pairs and writes them to one or more files (based on the size of the output).

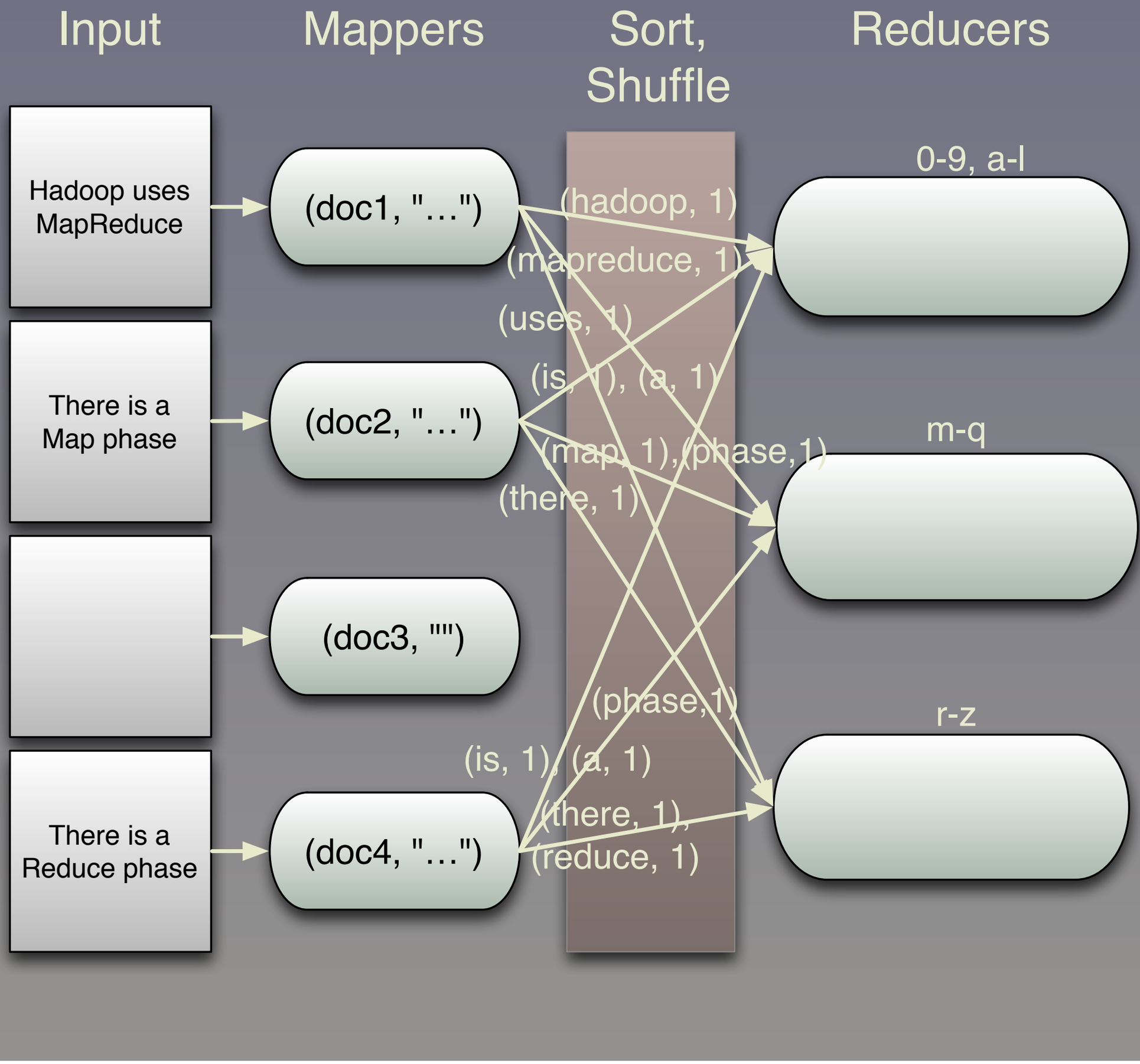


Each document gets a mapper. All data is organized into key-value pairs. We don't care about the key. The value is all the content or line by line. (Doesn't matter.) I'm showing the document contents in the boxes for this example. Actually, large documents might get split to several mappers (as we'll see). It is also possible to concatenate many small documents into a single, larger document for input to a mapper.

Each mapper will tokenize the doc contents, convert all words to lower case and count them...

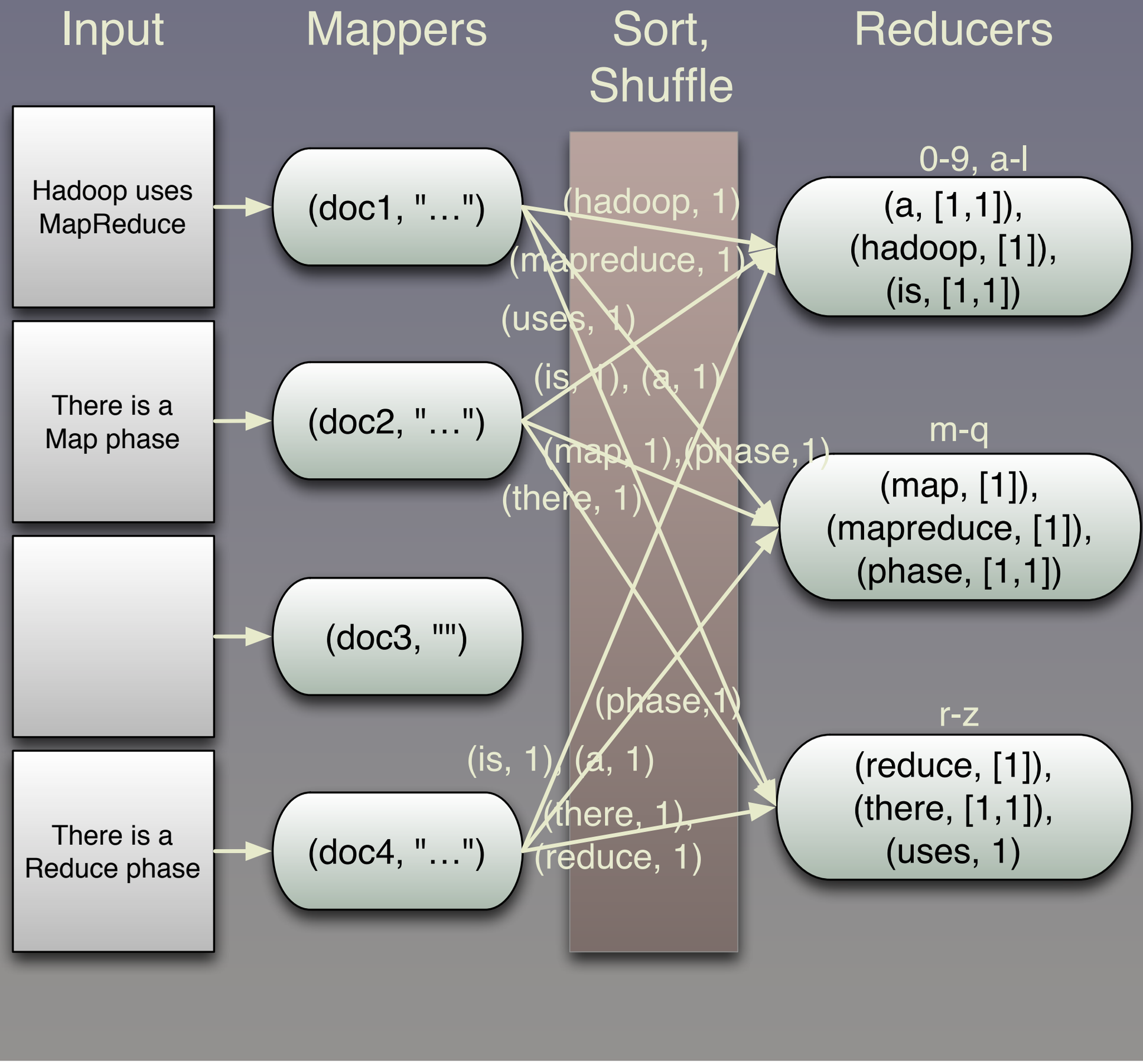


The mappers emit key-value pairs, where each key is one of the words, and the value is the count. In the most naive (but also most memory efficient) implementation, each mapper simply emits (word, 1) each time “word” is seen. However, this is IO inefficient!
Note that the mapper for the empty doc. emits no pairs, as you would expect.

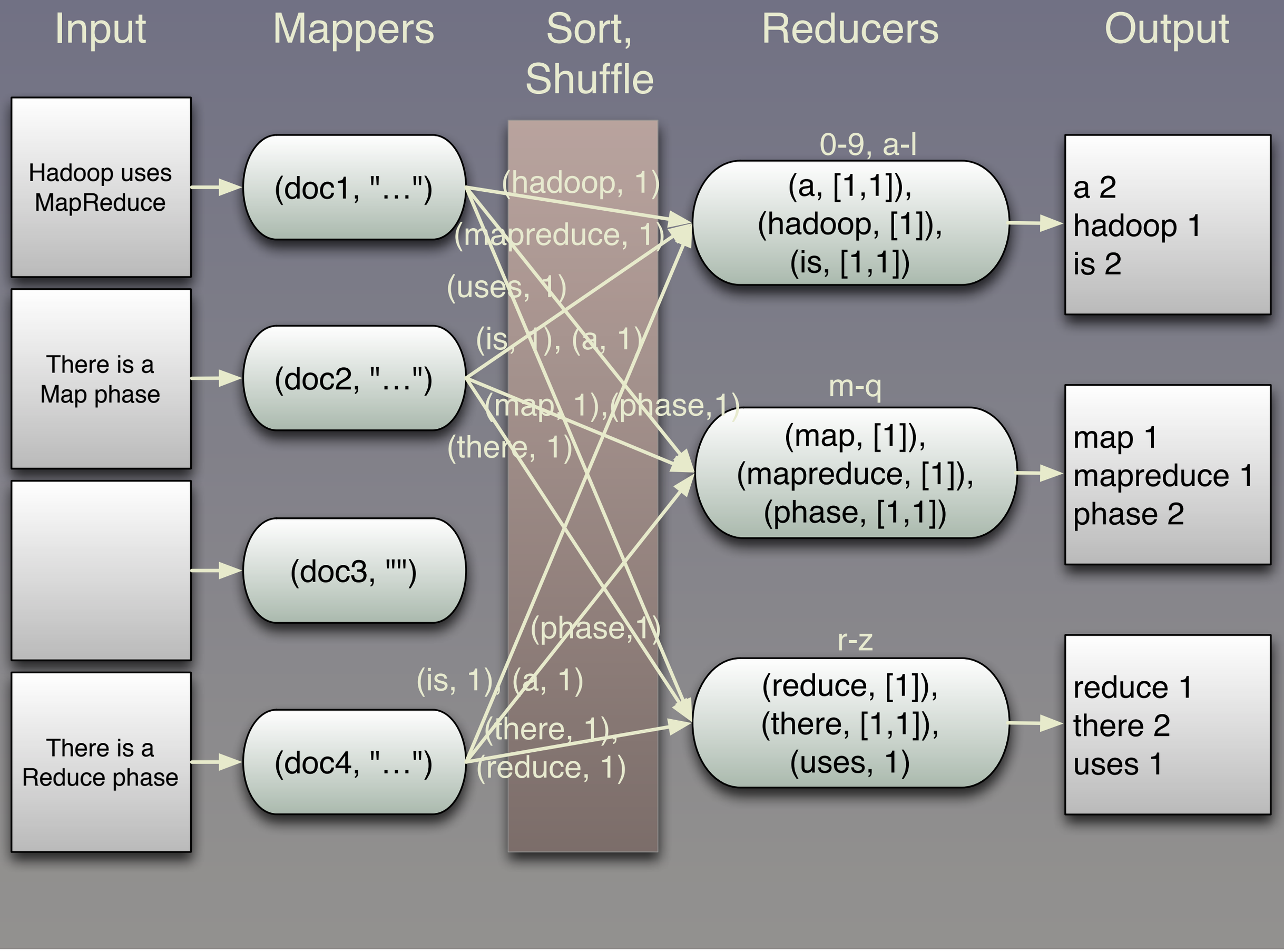


The mappers themselves don't decide to which reducer each pair should be sent. Rather, the job setup configures what to do and the Hadoop runtime enforces it during the Sort/Shuffle phase, where the key-value pairs in each mapper are sorted by key (that is locally, not globally) and then the pairs are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

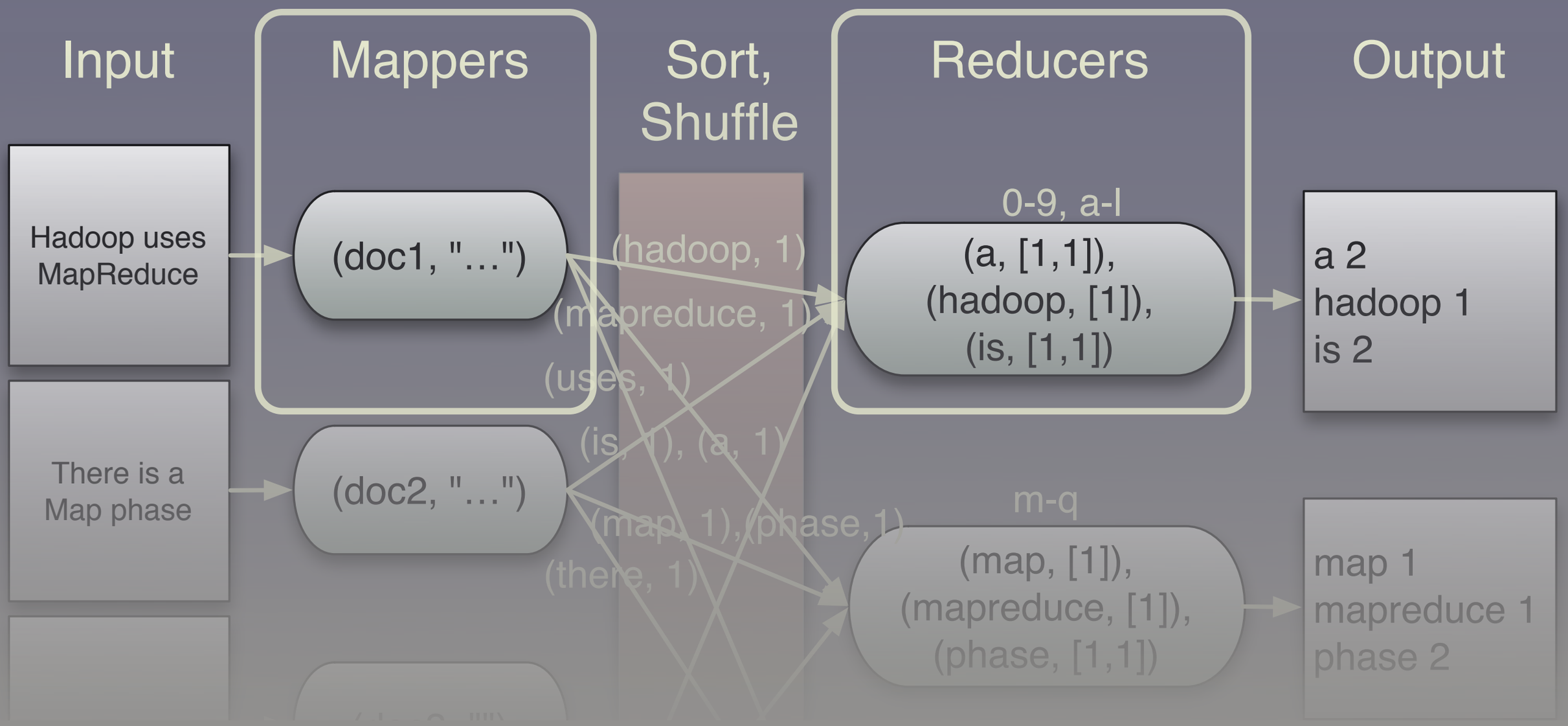
Note how we partitioned the reducers, by first letter of the keys. (By default, MR just hashes the keys and distributes them modulo # of reducers.)



Our reducers are passed each key and a collection of all the values for that key. (The MR framework creates this collection for us.)



The final view of the WordCount process flow. The reducer just sums the counts and writes the output. 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, but tab is common. (We'll discuss options as we go.)



Map:

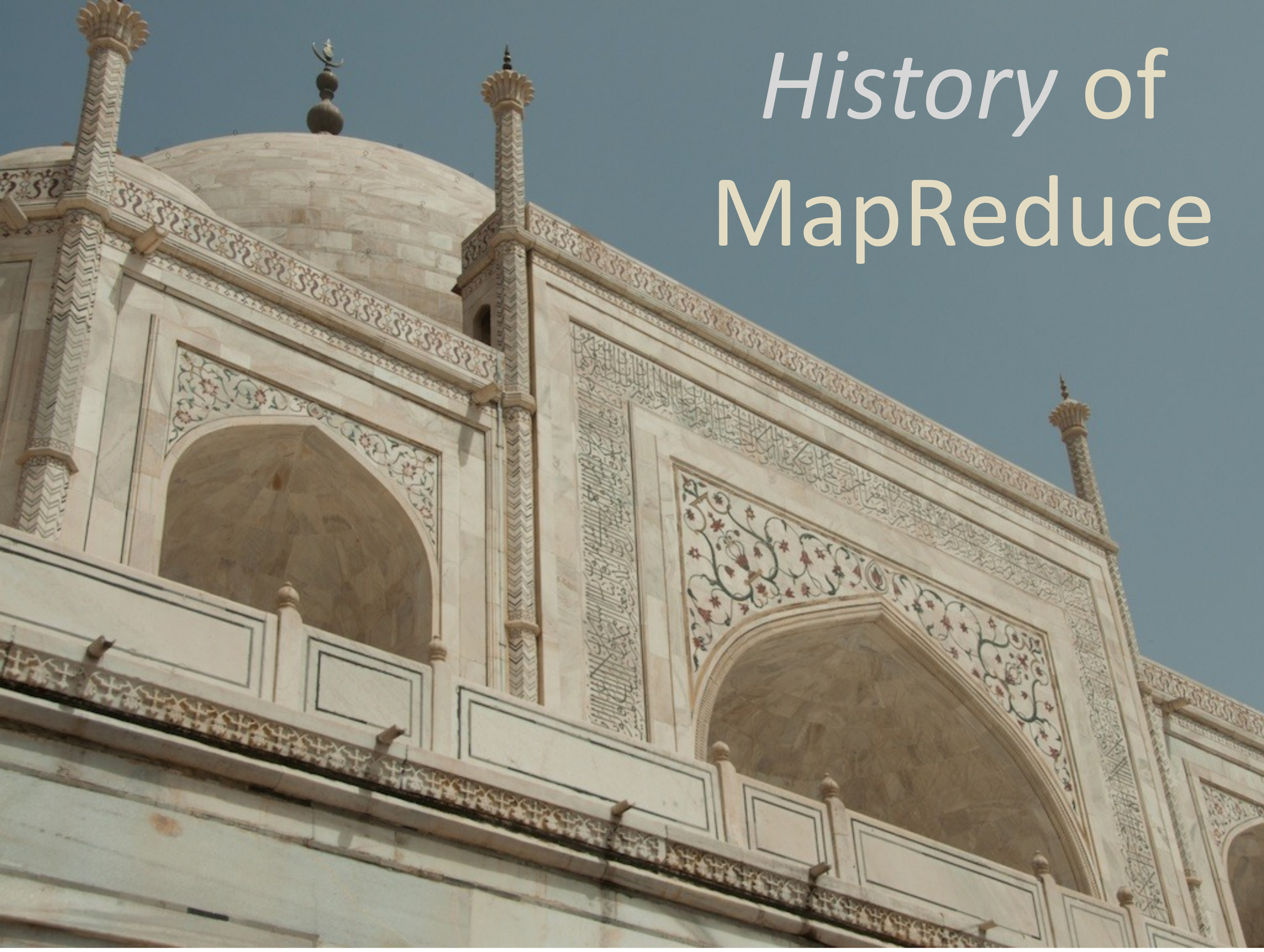
- Transform *one* input to *0-N* outputs.

Reduce:

- Collect *multiple* inputs into *one* output.

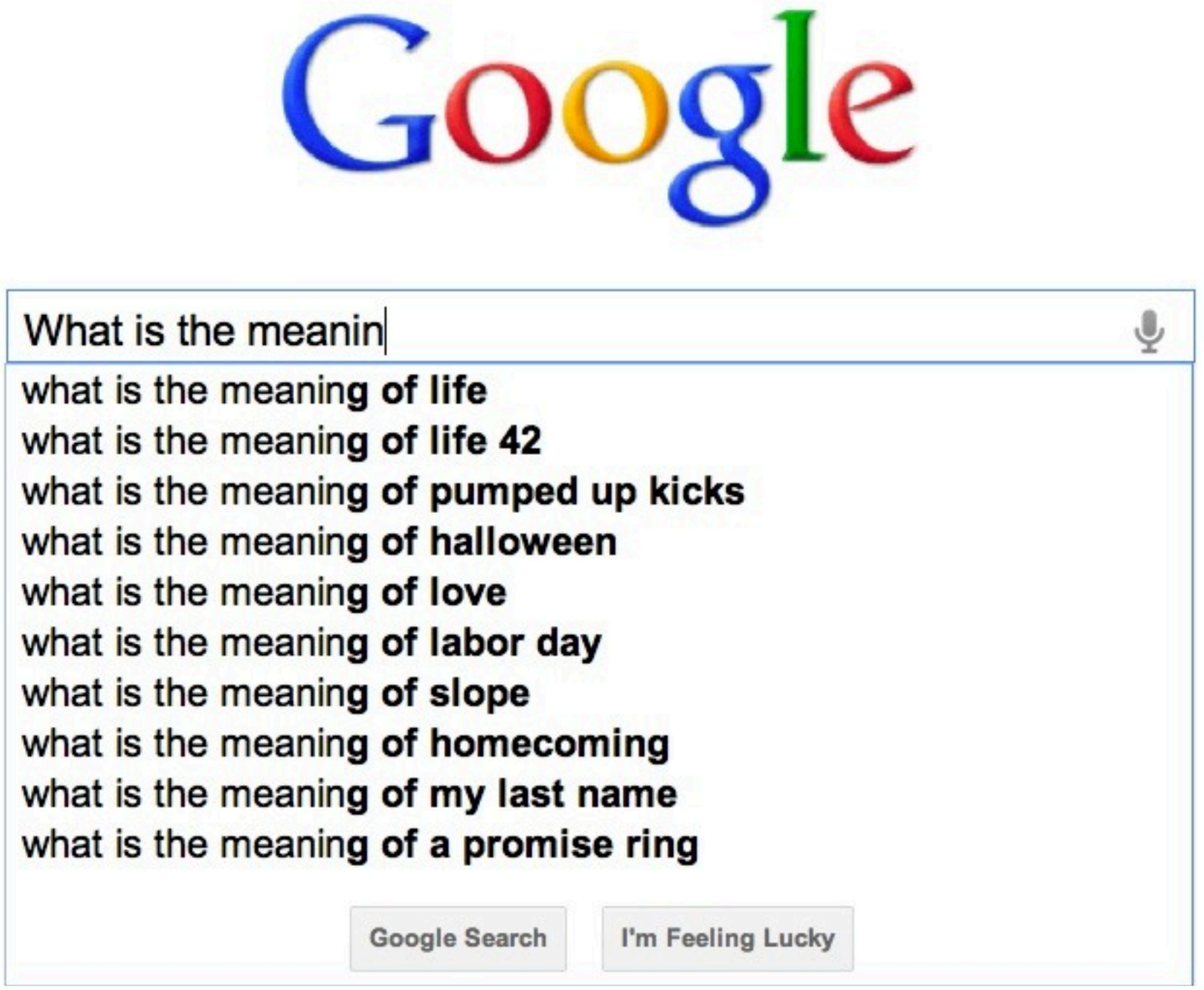
To recap, a “map” transforms one input to one output, but this is generalized in MapReduce to be one to 0-N. The output key-value pairs are distributed to reducers. The “reduce” collects together multiple inputs with the same key into

History of MapReduce

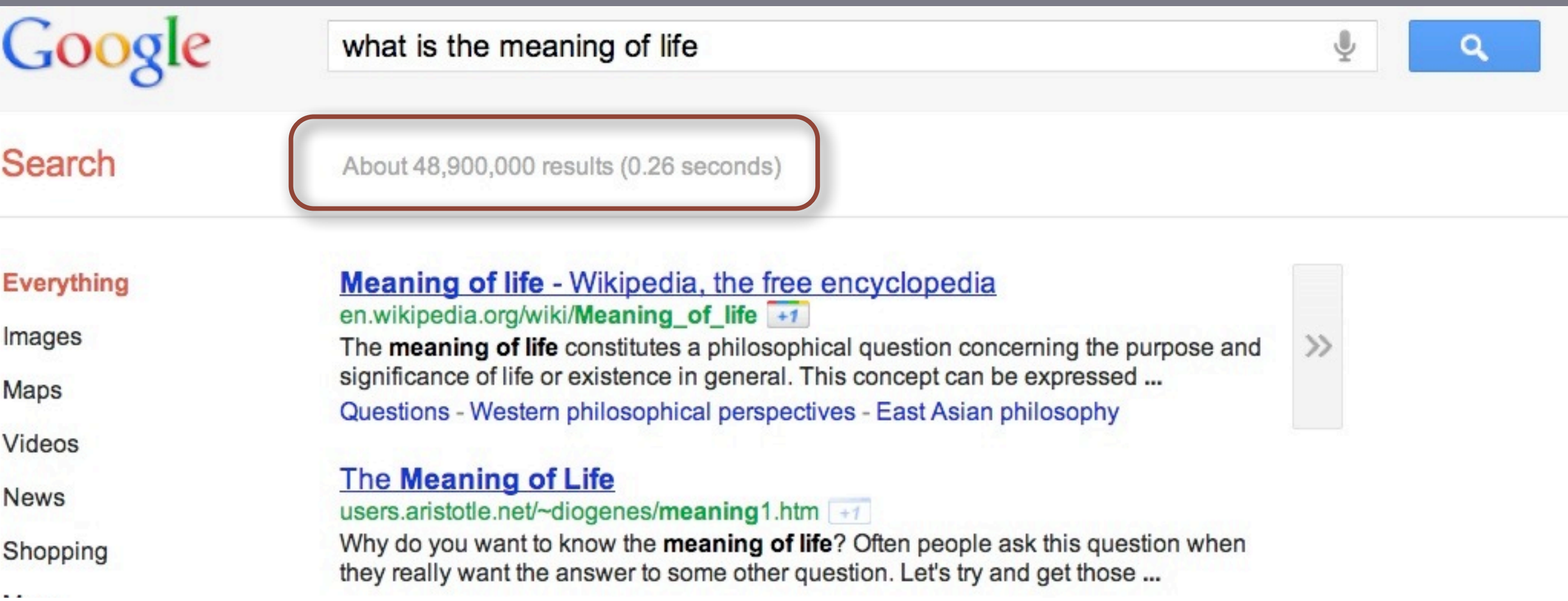


Let's review where MapReduce came from and its best-known, open-source incarnation, Hadoop.

How would you *index* the *web*?



How would you *index* the *web*?



You ask a *phrase* and the *search engine* finds the *best match* in *billions* of web pages.

Actually, *Google* computes the *index* that *maps* terms to pages in *advance*.

Google's famous *Page Rank* algorithm.

Google File System
for Storage

2003

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological envi-

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier

Tuesday, June 19, 12

A distributed file system provides horizontal scalability and resiliency when file blocks are duplicated around the cluster.

MapReduce for Computation

2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to par-

Tuesday, June 19, 12

A distributed file system provides horizontal scalability and resiliency when file blocks are duplicated around the cluster.

About this time, *Doug Cutting*, the creator of *Lucene* was working on *Nutch*...

He implemented clean-room versions of *MapReduce* and *GFS*...

By *2006* , they became part of a separate *Apache* project, called *Hadoop*.

The name comes from a toy, stuffed elephant that Cutting's son owned at the time.

Benefits of *MapReduce*

The best way to approach *Big Data* is to scale *Horizontally*.

MapReduce Design Goals

Maximize Disk I/O!!!

Oh, and run on *server-class*, *commodity* hardware.

Great for *Batch Mode*
Data Crunching.

MapReduce and Its *Discontents*

Is MapReduce the end of the story? Does it meet all our needs?

#1

The *Hadoop Java API* is too *verbose* *& object oriented.*

Even word count is not “obvious”. When you get to fancier stuff like joins, group-bys, etc., the mapping is not trivial at all.

#2

It's hard *implementing* many *Algorithms* in *MapReduce*.

Even word count is not "obvious". When you get to fancier stuff like joins, group-bys, etc., the mapping is not trivial at all.

#3

Some algorithms have *suboptimal performance* in MapReduce.

For example, *Graph algorithms*.

Tuesday, June 19, 12

While general purpose, some algorithms are not as fast as they could be, such as graph traversal algorithms (like Page Rank!!), because typically each traversal of an “arc” is a computation step. Hadoop is bad because every such ‘step’ is a separate process with a new JVM instance (by default). Also there is a fair amount of overhead moving data between mappers and reducers. (Google’s implementation may be more efficient.)

#4

What about *event processing* (“*real-time*”)?

Even Google now does not want to wait to compute a new index of the web. As soon as its web crawlers detect changes, Google would like to update its index as soon as possible.

Let's explore these issues, starting with the Hadoop Java API, but I'll show you Scala code; see my GitHub scala-hadoop project for the full code examples.

Scala

```
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))
    }
}
```


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.
I'm using Scala to call the Java API. If this were Java code, it would be more verbose.

The API is *invasive*
and it *obscures*
the application *logic*...

A MapReduce *Assembly Language*.

There is a very poor “separation of concerns”

How can we do better?

Decouple *logic* from *infrastructure*.

There is a very poor “separation of concerns”

How can we do better?

Use *Functional Programming*!!

This is data we're talking about, so we should be using mathematics. FP is as close as programming comes to mathematics!

Using *Crunch* (Java)

(Addressing #1 and #2)

Let's start by looking at options that improve the "MapReduce experience", but still work within that framework, but with a better API. Also, these options address #2, the difficulty of translating algorithms to MapReduce.

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

Distributed Collections,
Parallel Operations,
and *Pipelines*.

The collections include unordered sets and maps. Operations on these collections can be performed in parallel, including filtering, grouping, etc. Pipelines abstract over data processing flows. Note it was necessary to implement functional collection abstractions missing from Java.

Java

```
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;
  }
}
```

… still busy

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)

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.

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

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

    val pipeline = new Pipeline[WordCountExample]
    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))
  }
}
```

Scala

Is nice!

Tuesday, June 19, 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.

A *DSL* of *mathematical* operations.

Domain-Specific Language that hides the “assembly language” and exposes the most appropriate operations for working with data, various mathematical abstractions...

Data ⇔ Mathematics

∴ use *functional*

programming!

… and functional programming models mathematical operations ...

Data ⇔ Mathematics

e.g., map, flat map, fold, reduce, group by...

… such as map, flatmap, fold, reduce, …
Keep in mind that “MapReduce” originated from functional concepts, just bloated up...

There are other
imperative-OO flaws
in Hadoop...

Like reuse of objects (premature optimization), few natural key-value types (e.g., no tuple type available). I see a lot of Java bloat and poor modularity that make extensions and optimizations hard to implement.

Cascading (http://www.cascading.org/) is actually the best known, high-level Java API for Hadoop. However, using the Cascalog (https://github.com/nathanmarz/cascalog) or Scalding (https://github.com/twitter/scalding) wrappers in Clojure or Scala, respectively, provides the same benefits of lower boilerplate and more natural functional idioms for data.
Scoobi (https://github.com/nicta/scoobi) has many similarities to Scrunch.

Hive and Pig are popular parts of the Hadoop ecosystem, but they are limited. Hive is a SQL query language, which is actually pretty impressive, but not full-featured, Turing-complete language. Neither is Pig, which is a data flow language. (To be fair, both can be extended with code written in Java or other languages, but you avoid that with the Java/Clojure/Scala toolkits.)

Graph Processing

#3

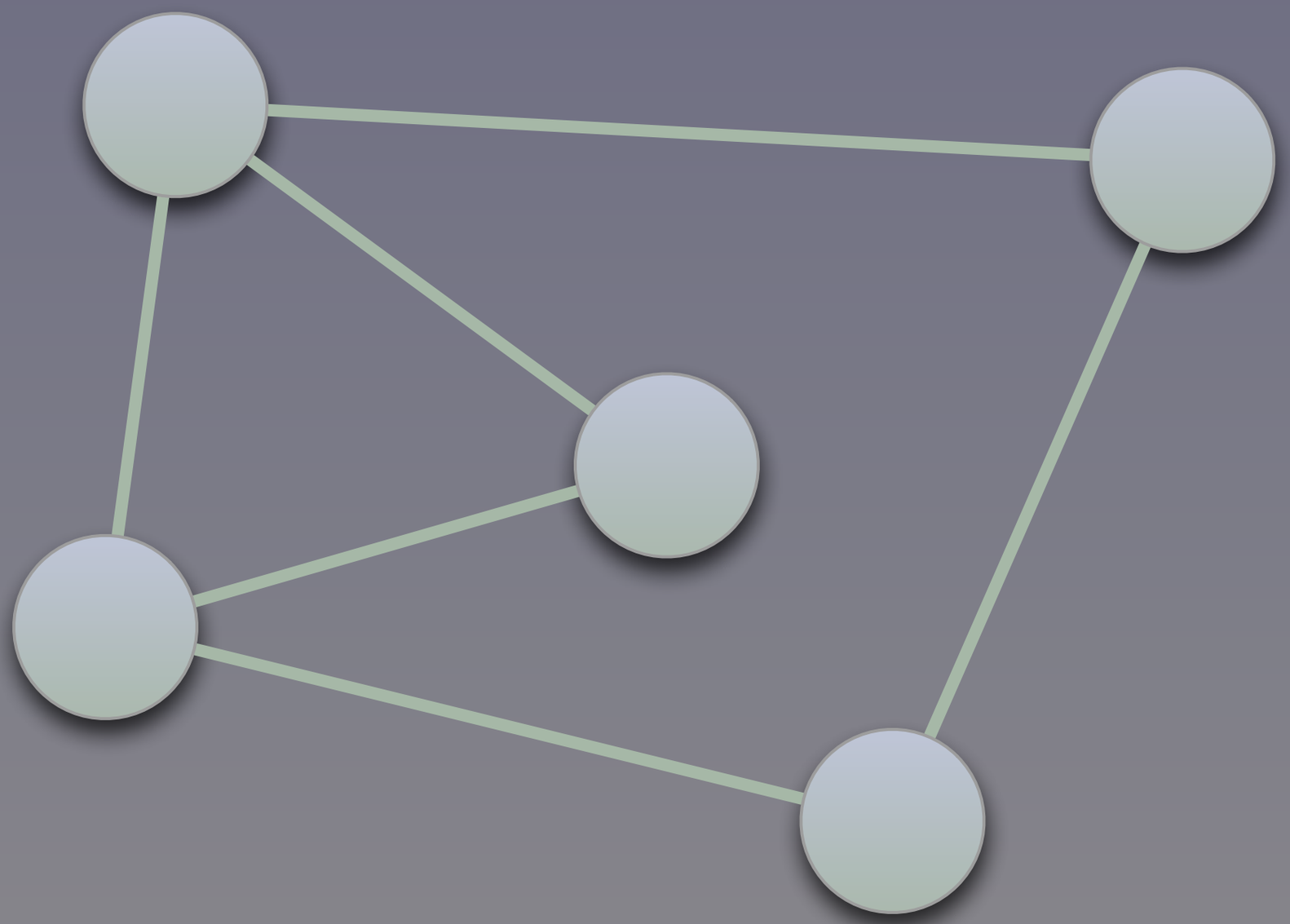
#3 is the issue of suboptimal performance, e.g., for graph processing, such as graph traversals, and discuss an alternative…
Examples of graphs include the Web itself, social networks, models of disease outbreaks, computer networks, highway maps, etc., etc.
Google extracts ~200 “features” from each web page it crawls.

Google is actually *moving* away from Page Rank on *MapReduce* to *Pregel*.

see http://googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google.html

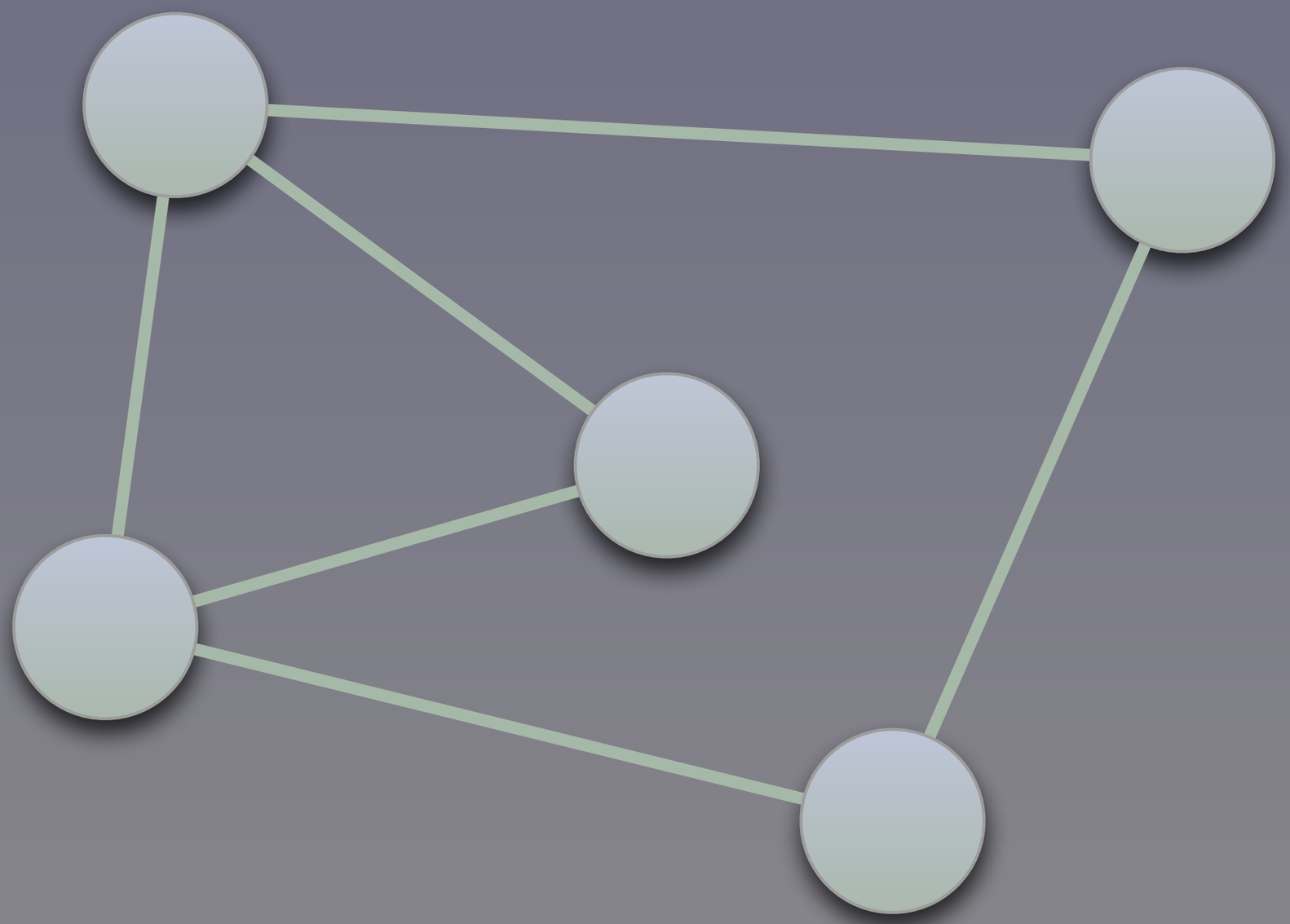
Pregel is inspired by *Bulk Synchronous Parallel* (BSP).

http://en.wikipedia.org/wiki/Bulk_Synchronous_Parallel

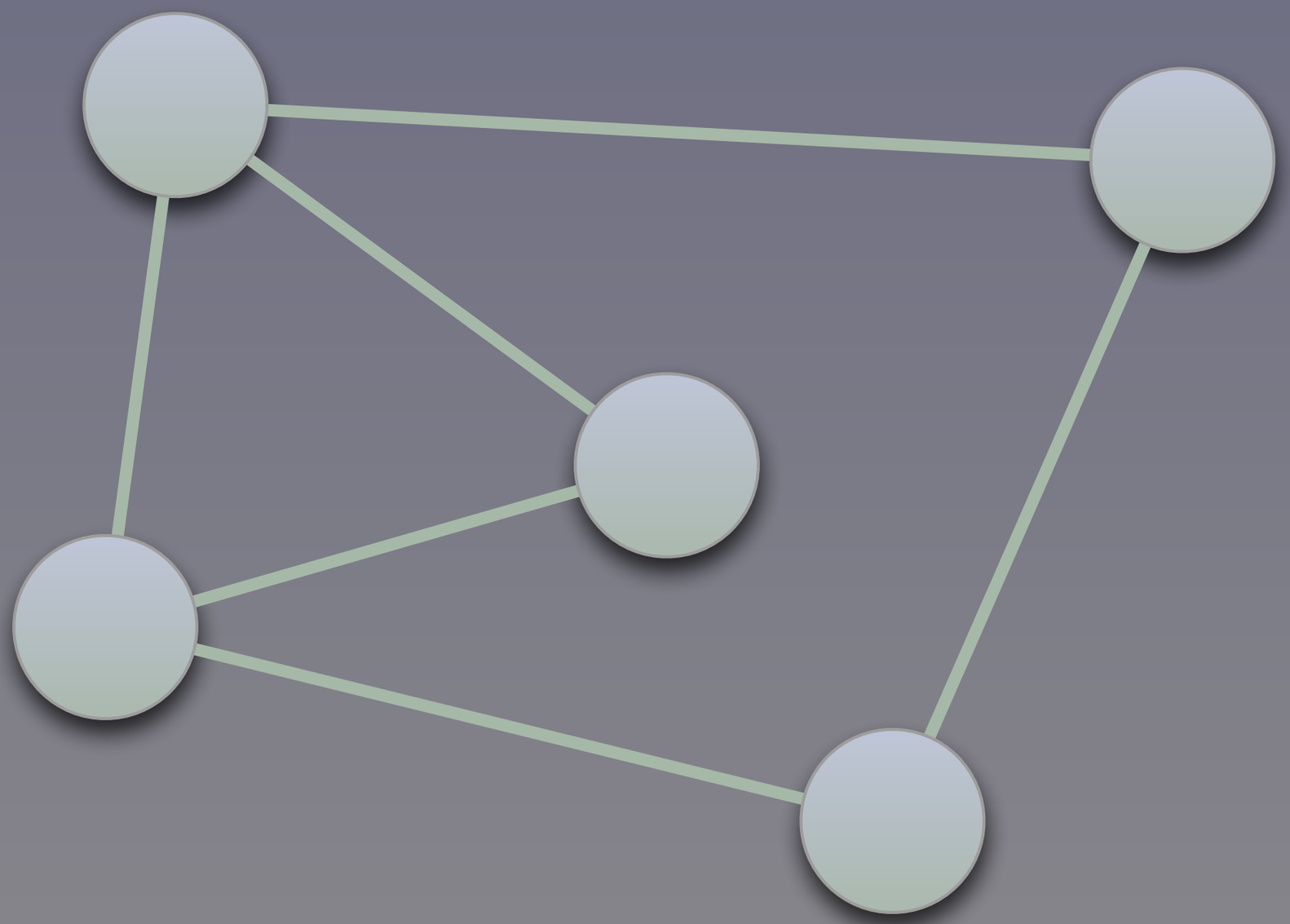


At time t_i, each node can: receive messages,

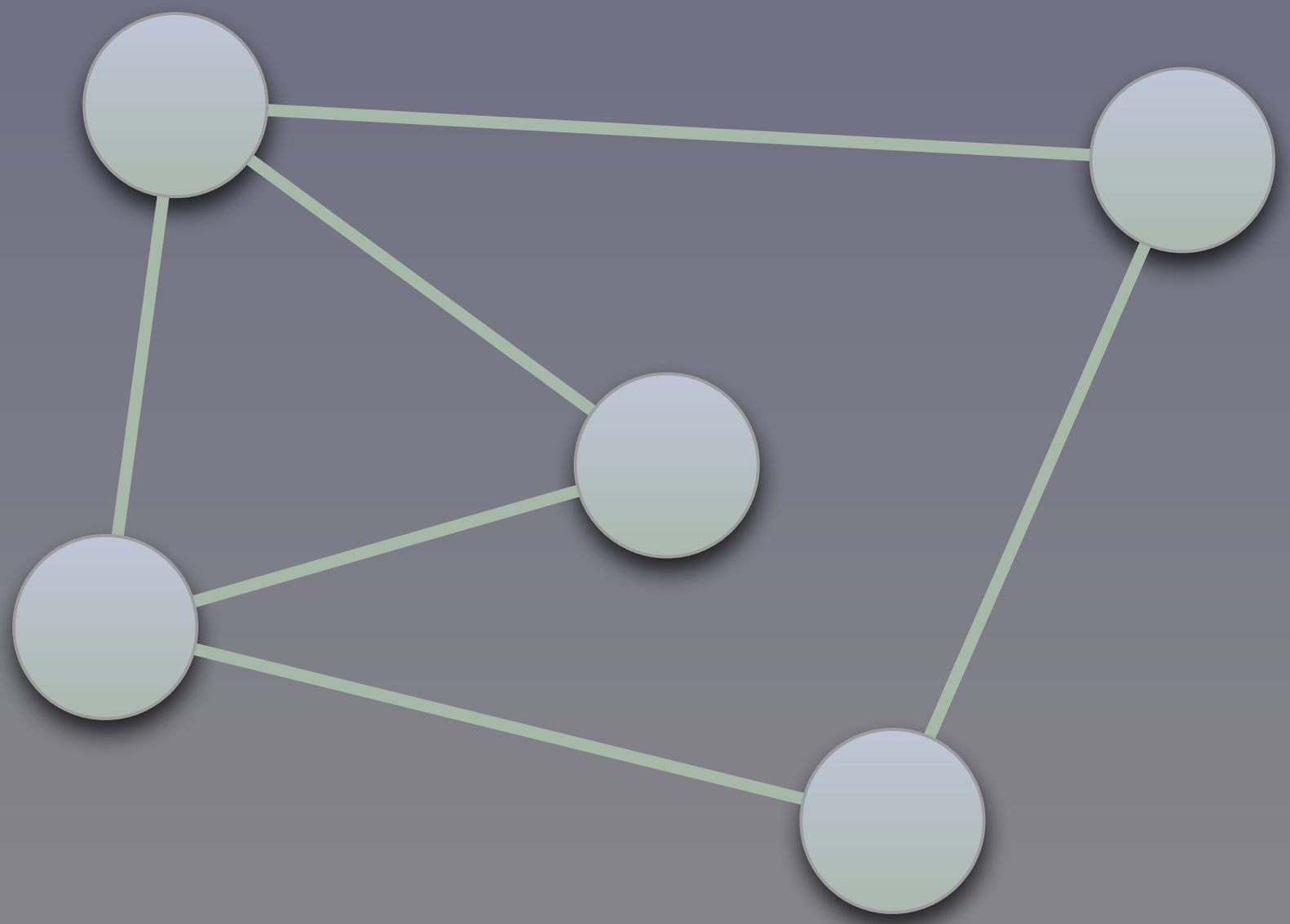
The system marches in synchronous steps, where processing happens at each time “ti”.



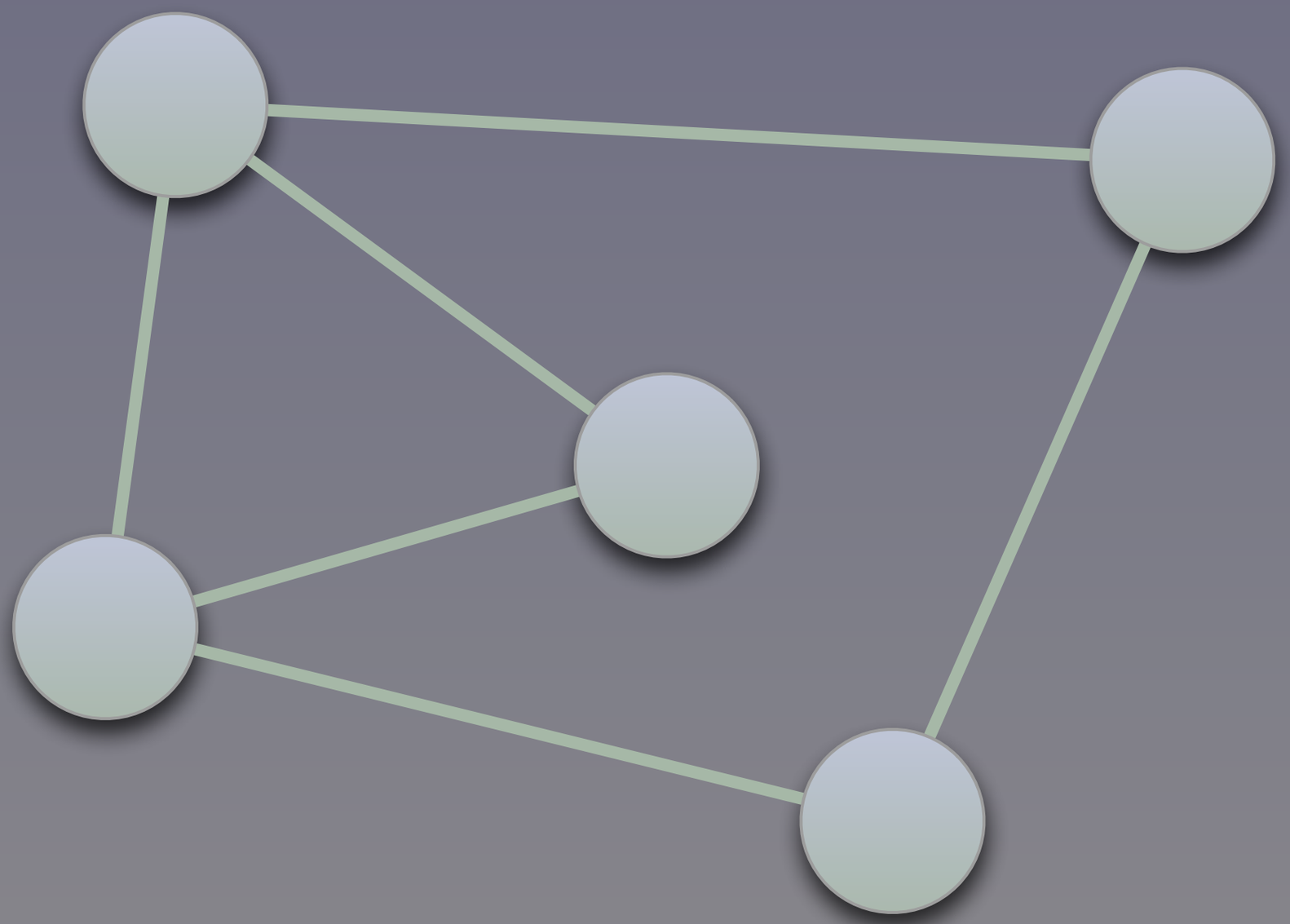
At time t_i, each node can: send messages,



At time t_i, each node can: modify its state,



At time t_i, each node can: modify the edges' states,



At time t_i, each node can:
mutate the graph topology.

The Apache Hama Project
http:// incubator.apache.org/

Apache > Apache Incubator > Hama Wiki | Blog | JIRA | SVN | Twitter | Last Published: 2012-03-06 | Version: 0.4.0-incubating

Information
- **Overview**
- Downloads
- Who We Are
- Privacy Policy

Resources
- Developers
- Mailing Lists
- Issue Tracking
- IRC channel

Documentation
- Getting Started
- Hama on Clouds
- Hama BSP tutorial
- Benchmarks
- Presentations
- Research Papers
- Books

Related Projects
- Hadoop

Apache Hama

Apache Hama is a pure BSP(Bulk Synchronous Parallel) computing framework on top of HDFS (Hadoop Distributed File System) for massive scientific computations such as matrix, graph and network algorithms. Currently, it has the following features:

- Job submission and management interface.
- Multiple tasks per node.
- Input/Output Formatter.
- Checkpoint recovery.
- Support to run in the Cloud using Apache Whirr.
- Support to run with Hadoop YARN.

Apache Giraph

Apache > Apache Incubator > Giraph > Welcome To Apache Incubator Giraph Wiki | JIRA | SVN | Last Published: 2012-06-06 | Version: 0.2-SNAPSHOT

Giraph
- **About**
- Wiki

Project Information
- Summary
- Team
- Mailing Lists
- License
- Issue Tracking
- Source Repository
- Dependencies
- Reports

Welcome To Apache Incubator Giraph

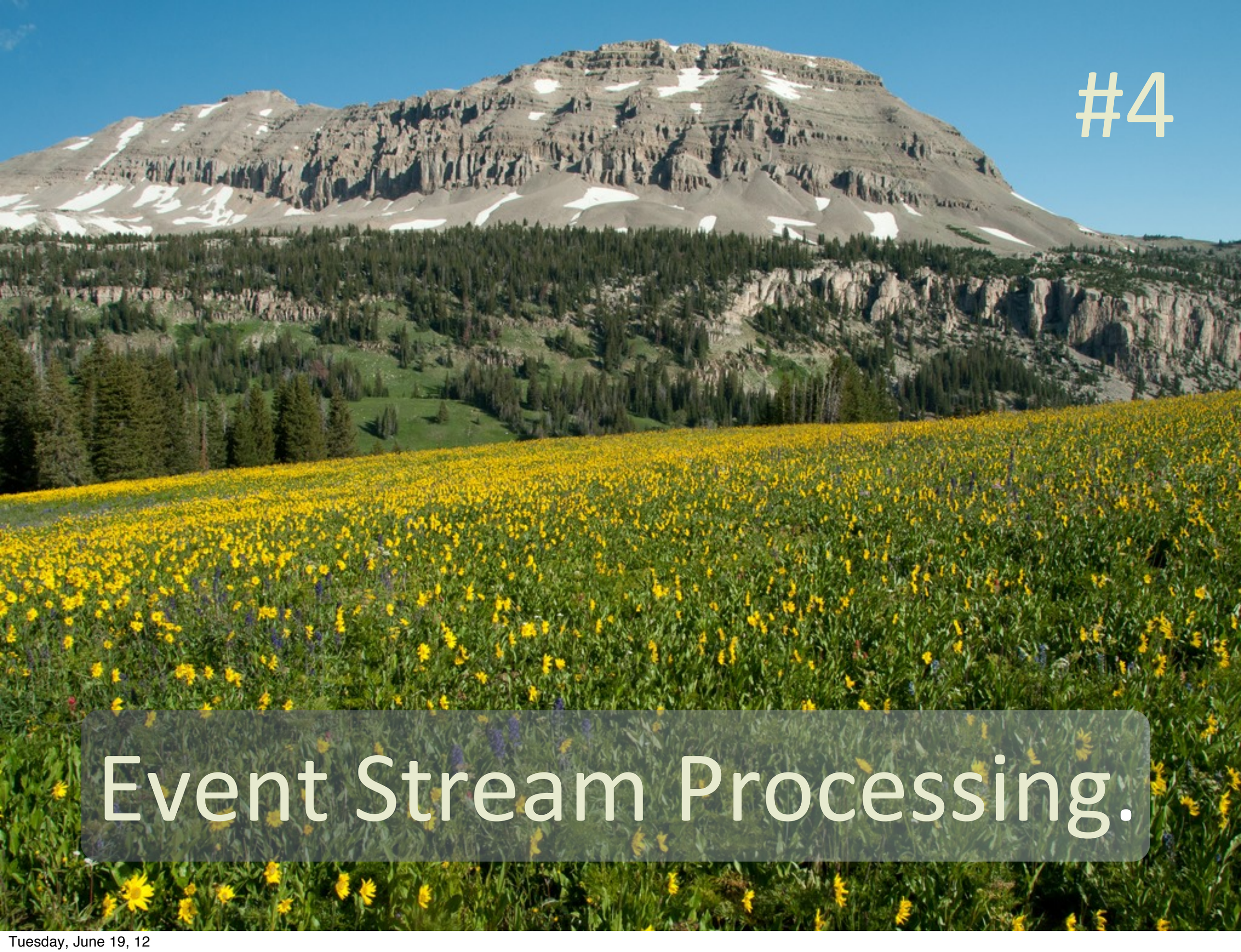
Web and online social graphs have been rapidly growing in size and scale during the past decade. In 2008, Google estimated that the number of web pages reached over a trillion. Online social networking and email sites, including Yahoo!, Google, Microsoft, Facebook, LinkedIn, and Twitter, have hundreds of millions of users and are expected to grow much more in the future. Processing these graphs plays a big role in relevant and personalized information for users, such as results from a search engine or news in an online social networking site.

Two BSP projects on Apache:
http://incubator.apache.org/hama/
http://incubator.apache.org/giraph/

Graph Databases

- Neo4J
- Allegro Graph
- FlockDB
- Titan
- GraphDB
- ...

http://www.neotechnology.com/
http://www.franz.com/agraph/allegrograph/
https://github.com/twitter/flockdb
http://thinkaurelius.github.com/titan/
http://www.sones.com/produkte (GraphDB)

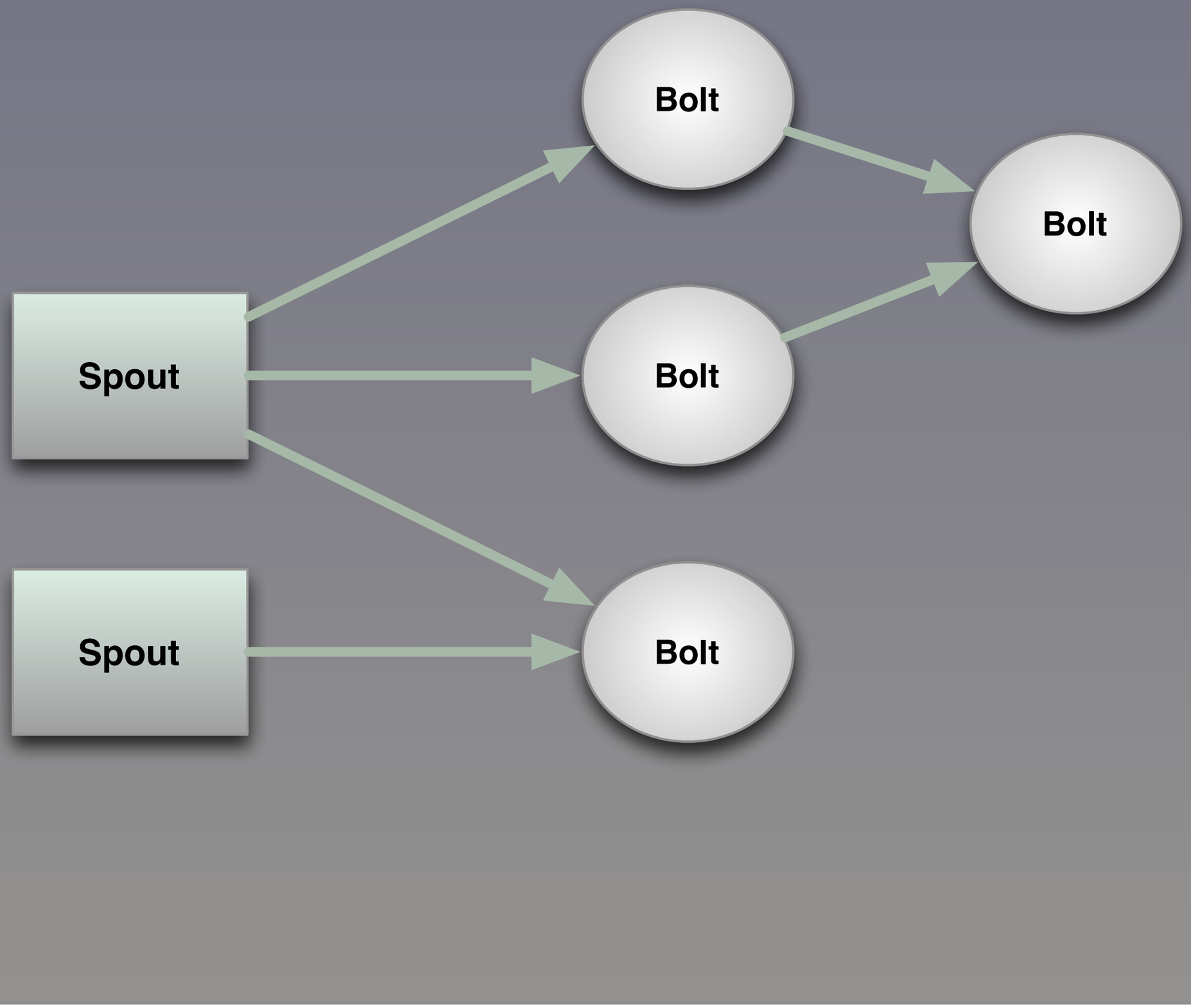


If you want to use Hadoop and process events in “real-time” (for some definition thereof), you’re best option is HBase, the Hadoop-oriented database. You could choose another NoSQL database too, depending on how much Hadoop interop you want. Hadoop’s MapReduce + HDFS itself are batch oriented and unacceptable. You might also want a dedicated event processing system...

Storm

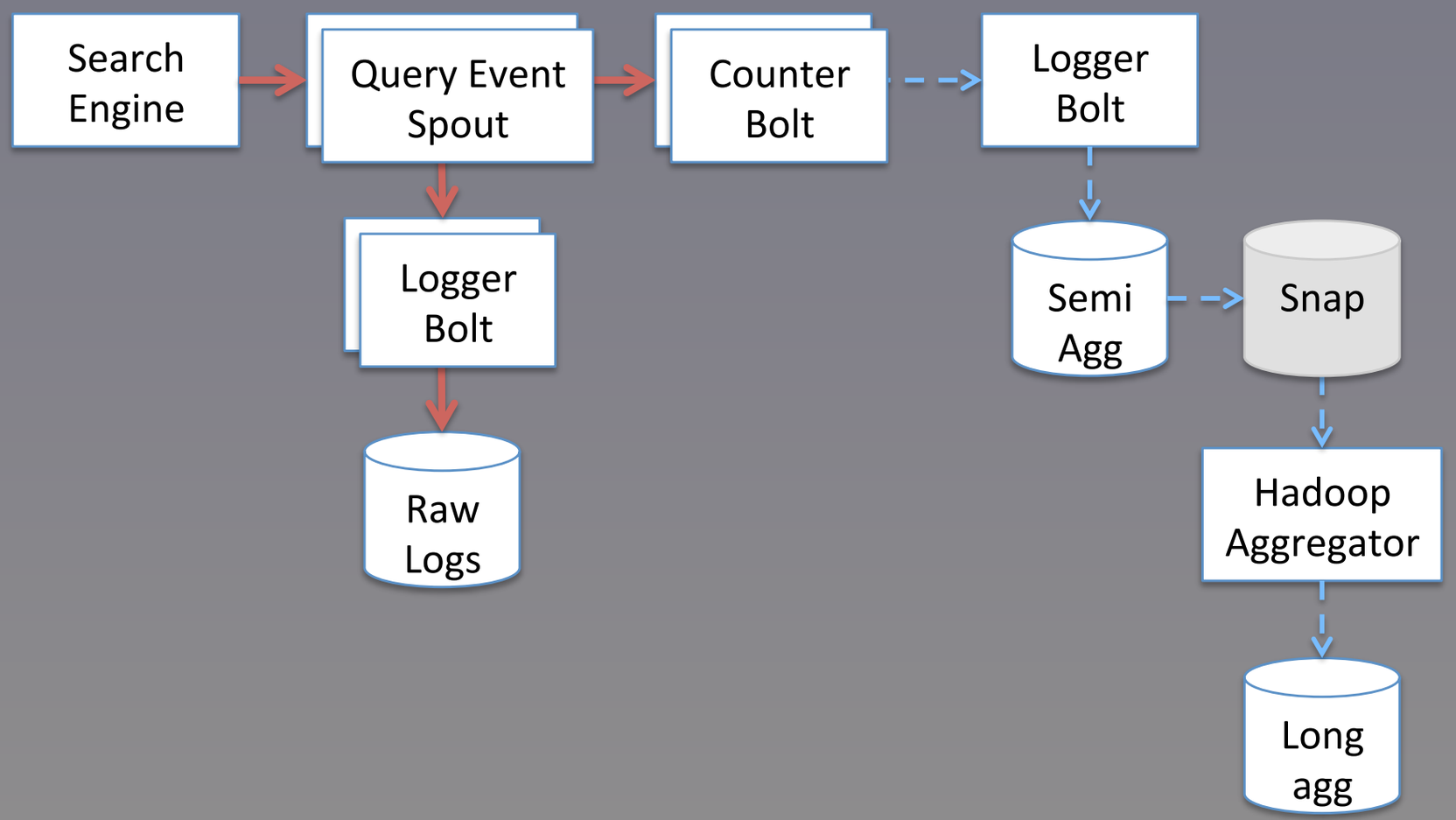
Distributed and fault-tolerant realtime computation

Taken from https://github.com/strangeloop/2011-slides/raw/master/Marz-Storm.pdf
See the project page, too: http://storm-project.net/



If you want to use Hadoop and process events in “real-time” (for some definition thereof), you’re best option is HBase, the Hadoop-oriented database. You could choose another NoSQL database too, depending on how much Hadoop interop you want. Hadoop’s MapReduce + HDFS itself are batch oriented and unacceptable. You might also want a dedicated event processing system...

Storm + Hadoop



You can combine Storm with Hadoop to get the best of both worlds.

A Manifesto...

So, I think we have an opportunity...

I worked with EJBs a decade ago. Like EJBs, Hadoop has an invasive API that obscures your business logic and reusability. There were too many configuration options in XML files. The framework “paradigm” is 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 and created large-scale, monolithic modules with few abstractions for extension and optimization points. I’ve also 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 FP community is well positioned to create the next generation.

V2.0 fixes *issues*, but it still feels like *J3EE*.

The “next gen.”, V2.0 Hadoop fixes some problems, but I think this first-generation infrastructure has too many flaws to be dominant for a long time (at least outside large enterprises that always stick with suboptimal solutions sold by big-name players). The Java-OO eccentricities, the overly-large and bloated modules, the premature optimization (and missing optimizations that wouldn’t be premature), lead me to believe that Hadoop will be displaced the same way that the Spring Framework displaced EJBs.

Stop using *Java*.

Seriously, it's the worst possible language for Big Data. It promotes bloat and bad abstractions. Without a REPL, you can't experiment. You are penalizing yourself by using it.
You don't have to rewrite your old code, you can just call it form Scala or Clojure, then use those languages (or pick anything else modern, like JRuby and Jython or a non-JVM, functional language).

Assess your *needs.*

Use optimal *tools.*

Hadoop and (most) of the ecosystem tools DO have their place, but understand your needs first and then pick the tools that make sense. Skepticism is healthy. If you decide you do need Hadoop scalability (and many people do), then use higher-level languages and tools like Cascading (or better, Cascalog or Scalding) or Hive and Pig. And don't assume you should use every Hadoop ecosystem tool; some are garbage.

Consider *Storm and BSP*

Storm provides “real-time”, event-stream processing and BSP (Bulk Synchronous Parallel) is one of several emerging, performant technologies for graph algorithms.

Questions?

QCon NYC
June 19, 2012

@deanwampler
thinkbiganalytics.com

THiNK BIG ANALYTICS