

GOTO Chicago 2013  
[dean@concurrentthought.com](mailto:dean@concurrentthought.com)  
[@deanwampler](https://twitter.com/deanwampler)  
[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)

# What's Ahead for Big Data?



Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

Copyright © Dean Wampler, 2011-2013, All Rights Reserved. Photos can only be used with permission. Otherwise, the content is free to use.

Photo: John Hancock Center, Michigan Ave.



# Dean Wampler...

Copyright © 2011-2015, Dean Wampler, All Rights Reserved

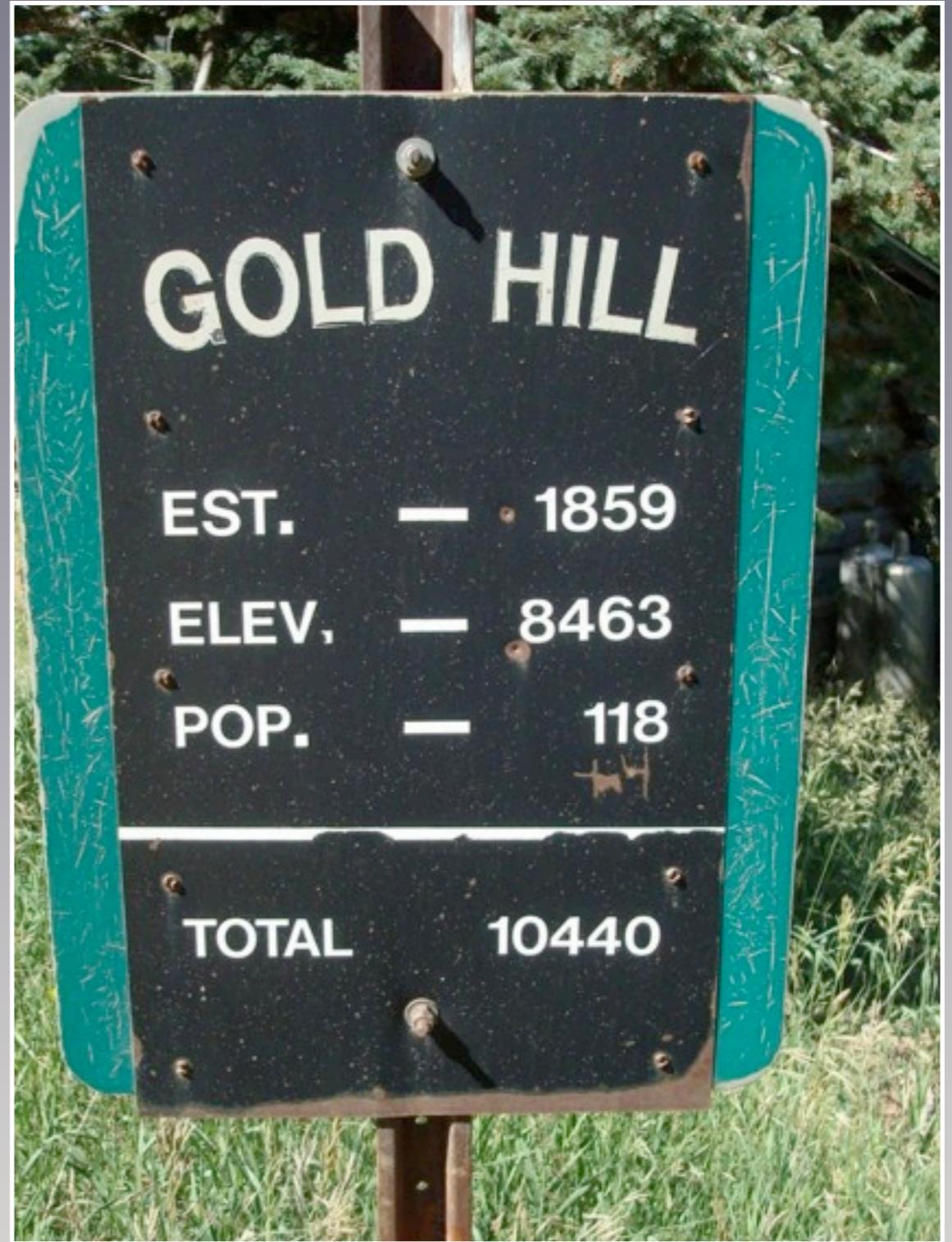
Tuesday, April 23, 13

My books...

Photo: The Chicago River, ~1.5 miles SW of here!

# 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



Copyright © 2011-2013, Dean Wampler. All Rights Reserved

Tuesday, April 23, 13

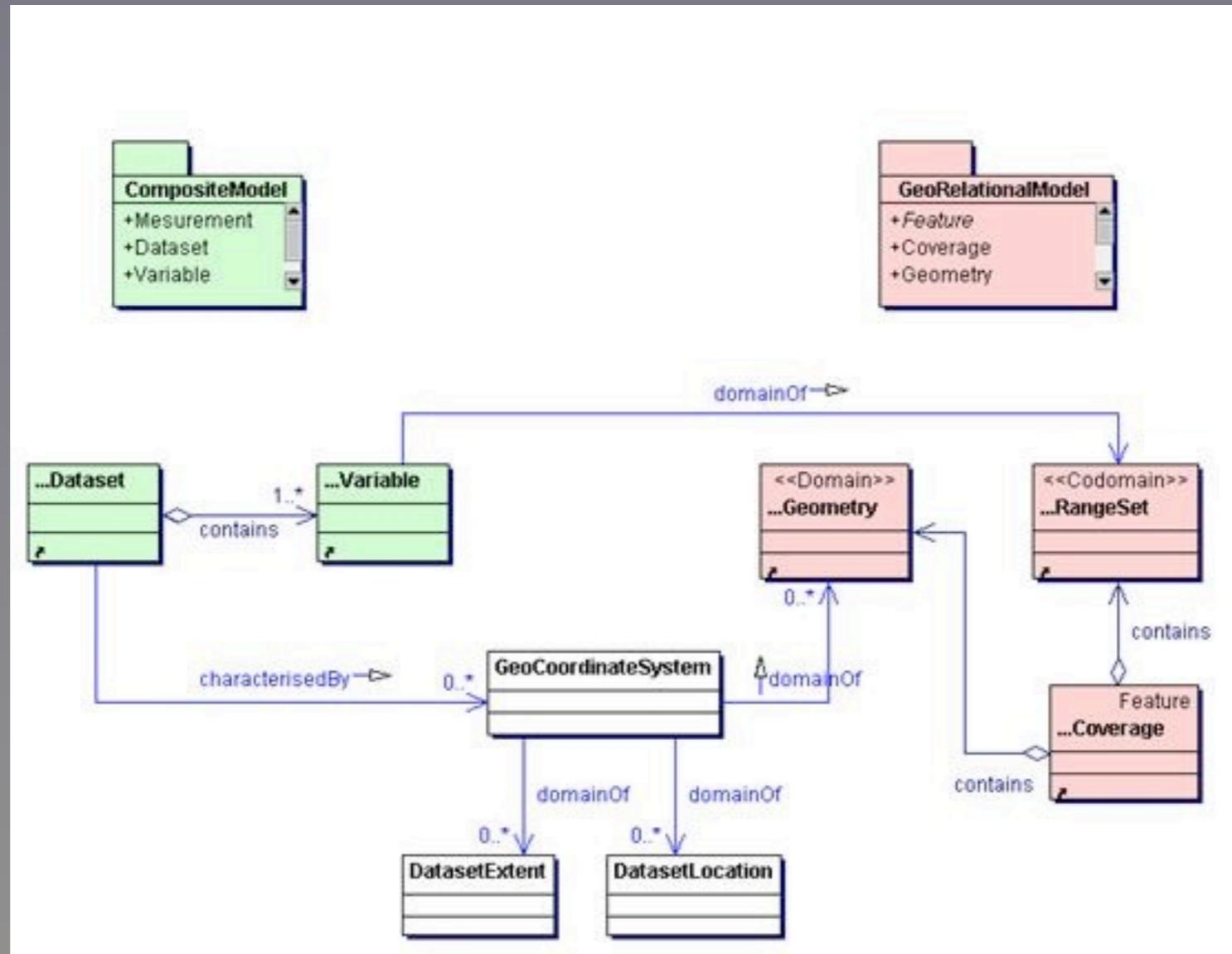
Three trends influence my thinking...

Photo: Prizker Pavilion, Millenium Park

# Data Size ↑



# 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, and we need to integrate so many disparate sources of data. So, using relatively-agnostic software, e.g., collections of things where the software is more agnostic about the structure of the data and the domain, tends to be faster to develop, test, and deploy. Put another way, we find it more useful to build somewhat agnostic applications and drive their behavior through data...

# Data-Driven Programs ↑



Tuesday, April 23, 13

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, as you might expect...)

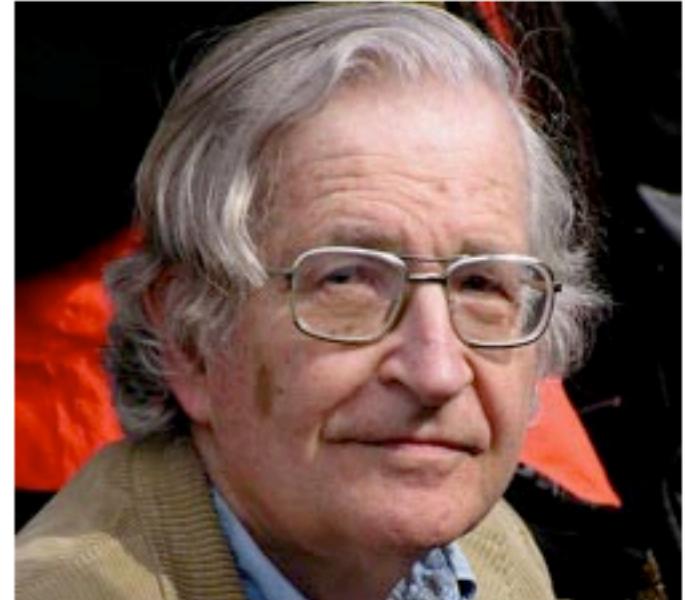
# Probabilistic Models vs. Formal Grammars

[tor.com/blogs/...](http://www.tor.com/blogs/)

## Norvig vs. Chomsky and the Fight for the Future of AI

KEVIN GOLD

When the Director of Research for Google compares one of the most highly regarded linguists of all time to Bill O'Reilly, you know it is *on*. Recently, Peter Norvig, Google's Director of Research and co-author of [the most popular artificial intelligence textbook in the world](#), wrote a [webpage](#) extensively criticizing Noam Chomsky, arguably the most influential linguist in the world. Their disagreement points to a revolution in artificial intelligence that, like many revolutions, threatens to destroy as much as it improves. Chomsky, one of the old guard, wishes for an elegant theory of intelligence and language that looks past human fallibility to try to see simple structure underneath. Norvig, meanwhile, represents the new philosophy: truth by statistics,



Chomsky photo by Duncan Rawlinson and his Online Photography School. Norvig photo by Peter Norvig

An interesting manifestation of this trend is the public argument between Noam Chomsky and Peter Norvig on the nature of language. Chomsky long ago proposed a hierarchical model of formal language grammars. Peter Norvig is a proponent of probabilistic models of language. Indeed all successful automated language processing systems are probabilistic.

<http://www.tor.com/blogs/2011/06/norvig-vs-chomsky-and-the-fight-for-the-future-of-ai>

# Big Data Architectures

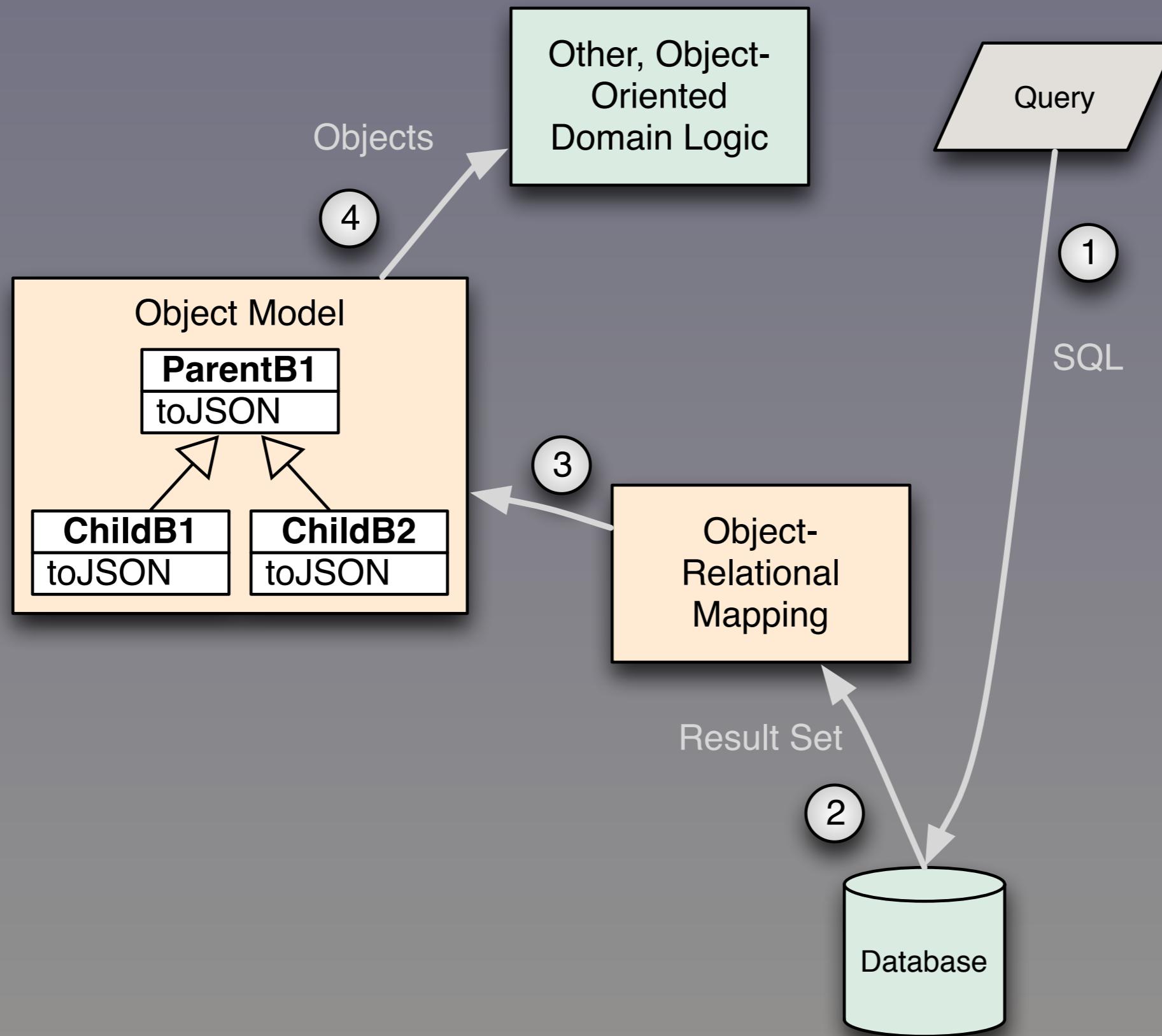


Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

What should software architectures look like for these kinds of systems?

Photo: Cloud Gate (a.k.a. "The Bean") in Millenium Park



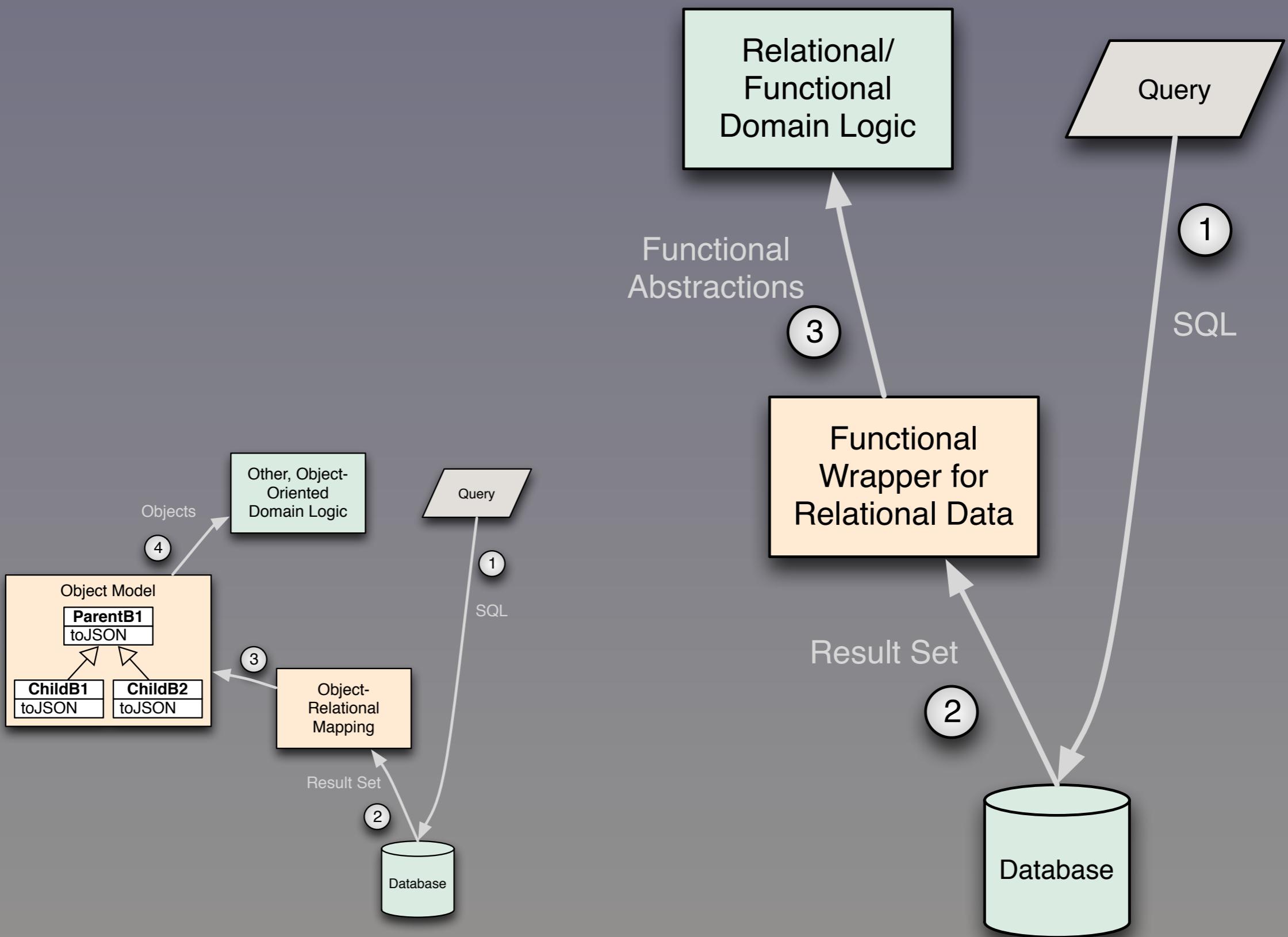
10

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

Traditionally, 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, 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...)

The **toJSON** methods are there because we often convert these object graphs back into fundamental structures, such as the maps and arrays of JSON so we can send them to the browser!



II

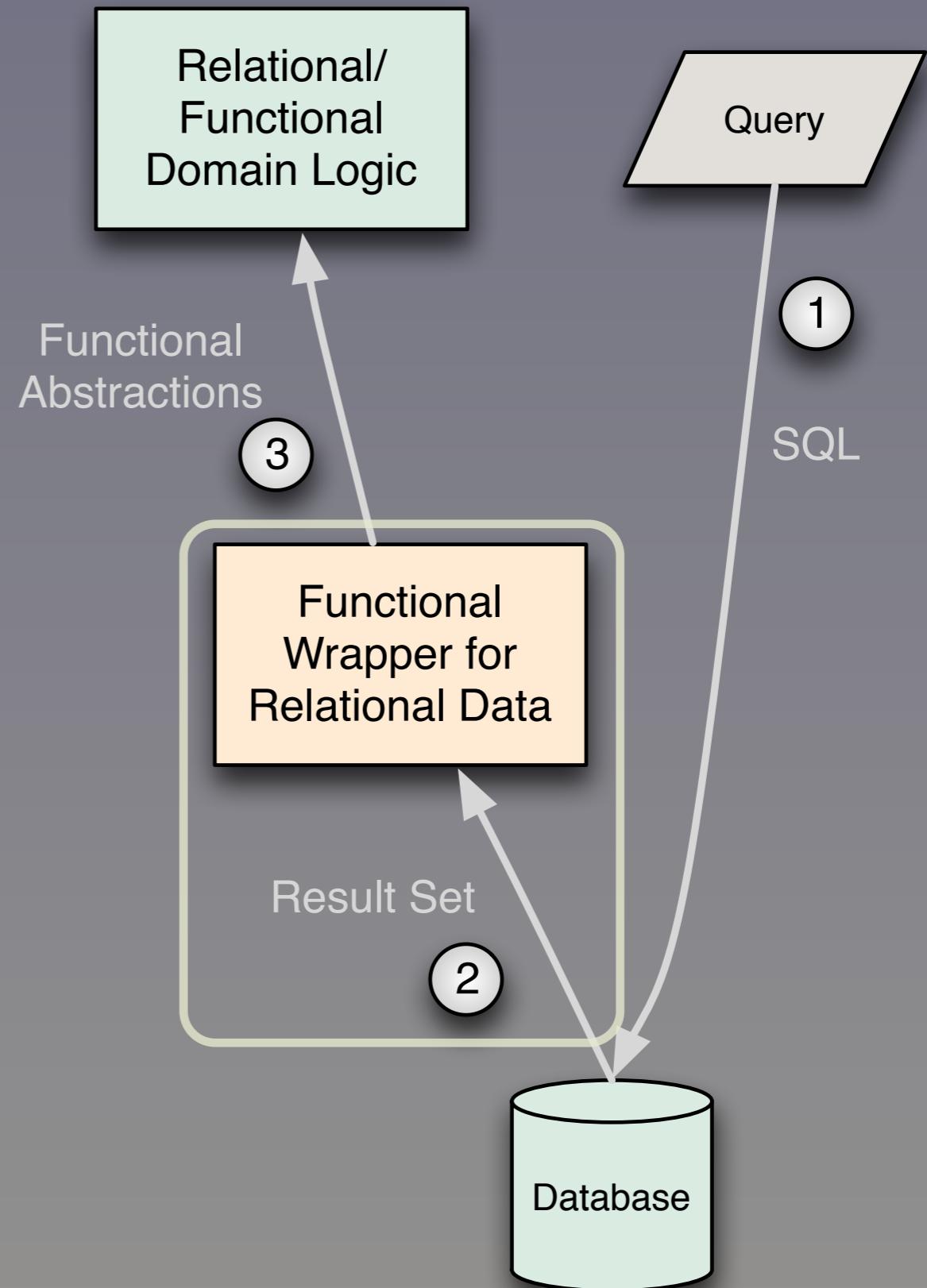
Copyright © 2011-2013, Dean Wampler, All Rights Reserved

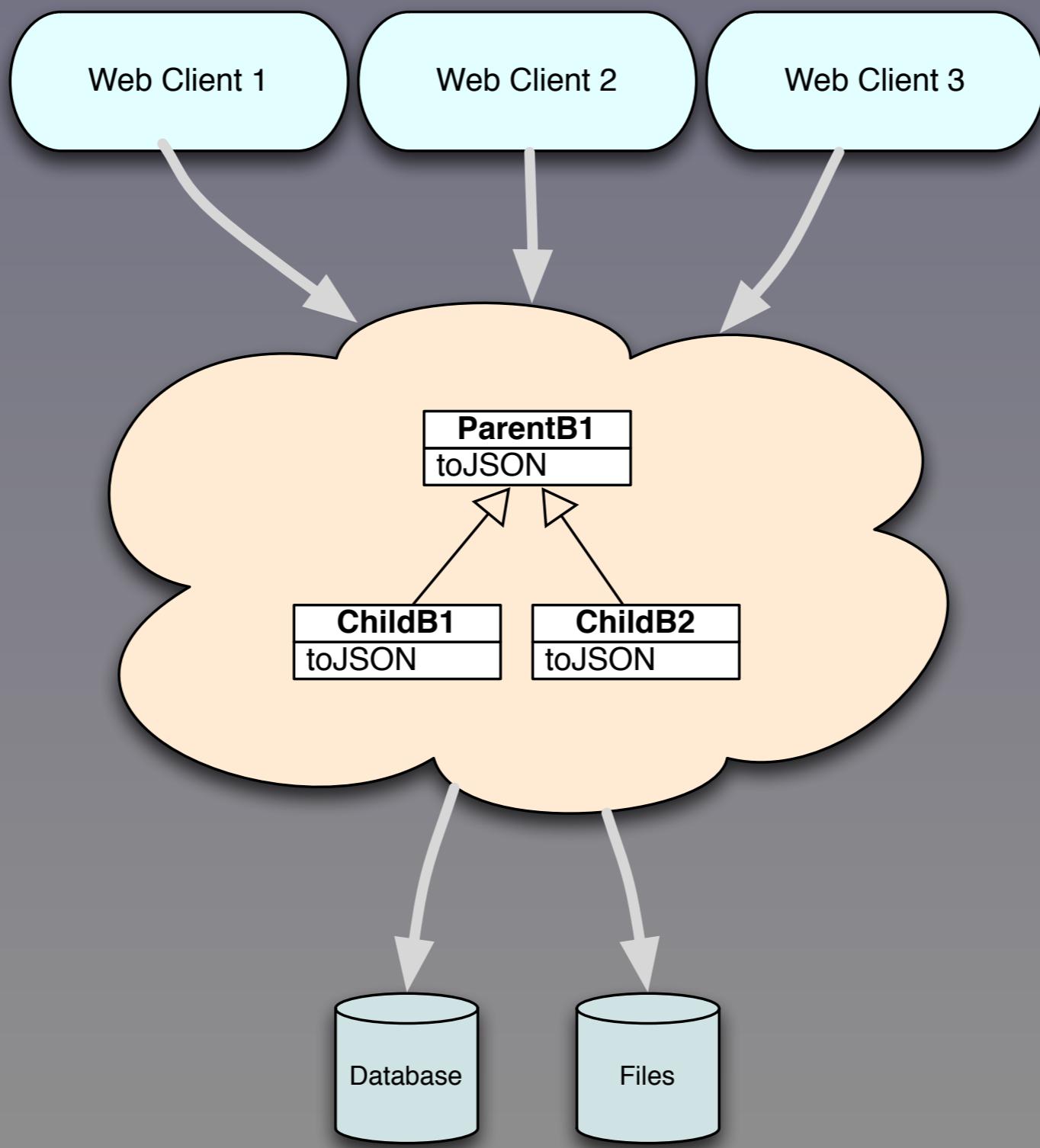
Tuesday, April 23, 13

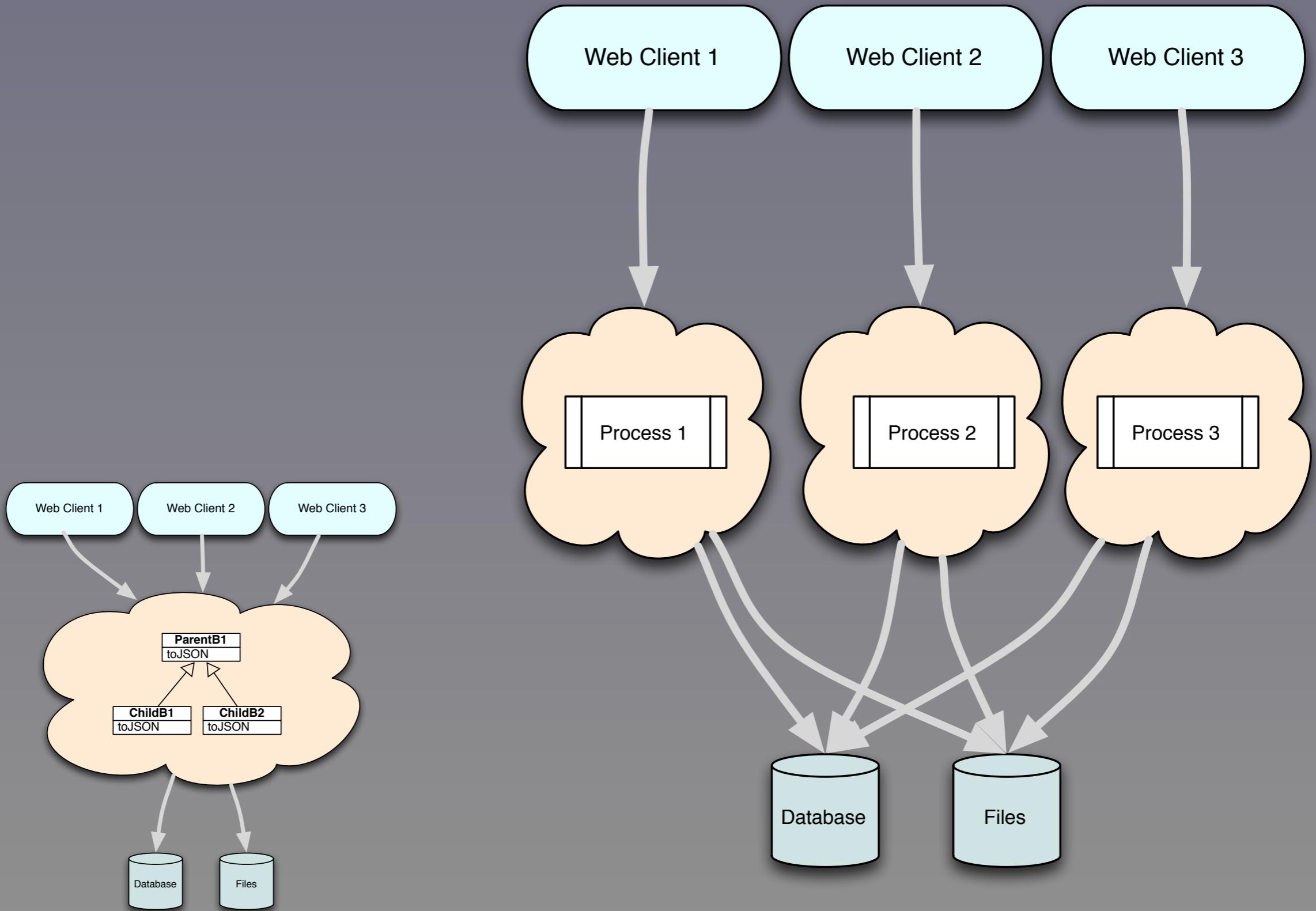
But the traditional systems are a poor fit for this new world: 1) they add too much overhead in computation (the ORM layer, etc.) and memory (to store the objects). Most of what we do with data is mathematical transformation, so we're far more productive (and runtime efficient) if we embrace fundamental data structures used throughout (lists, sets, maps, trees) and build rich transformations into those libraries, transformations that are composable to implement business logic.

- Focus on:

- Lists
- Maps
- Sets
- Trees
- ...





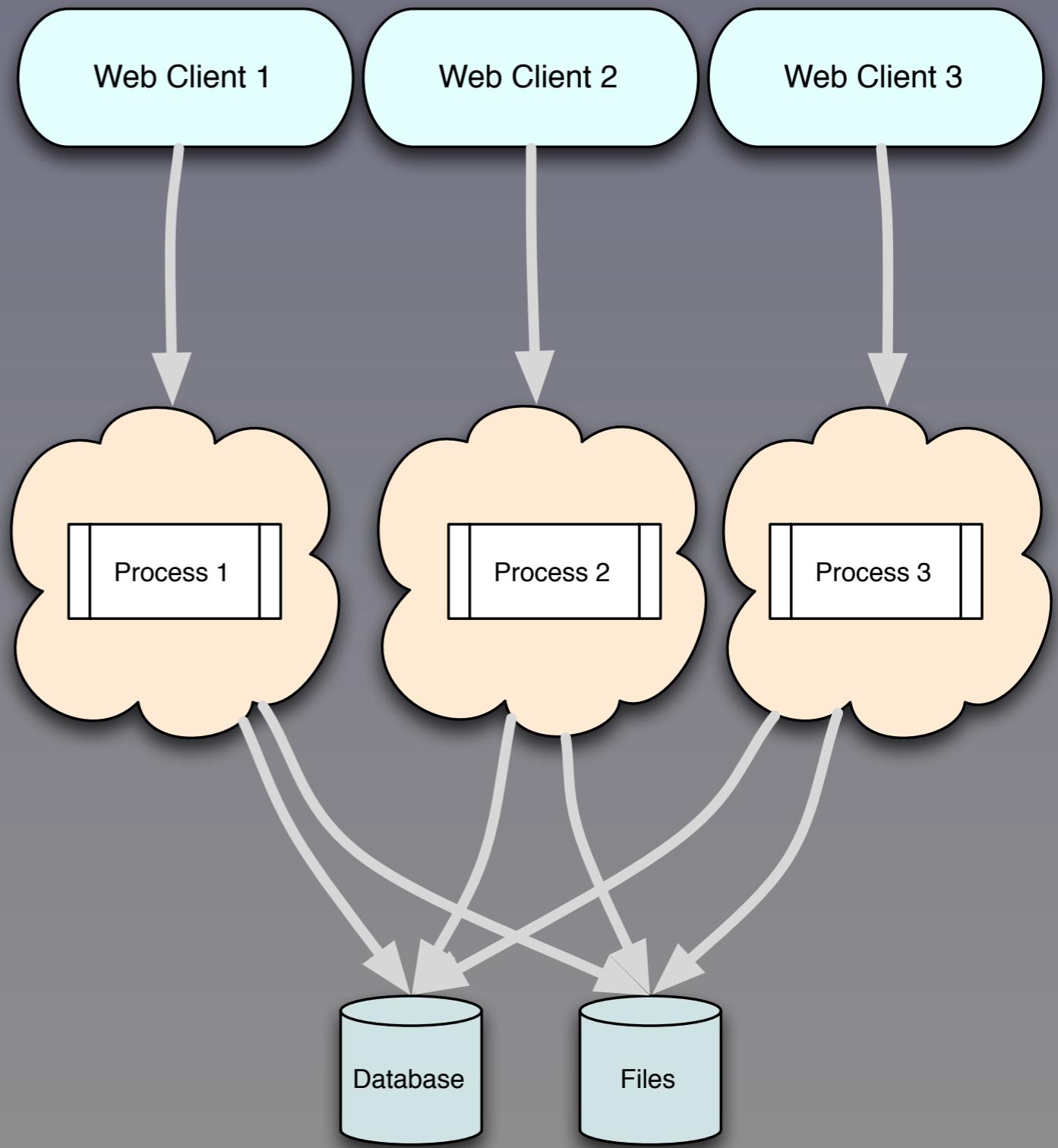


Tuesday, April 23, 13

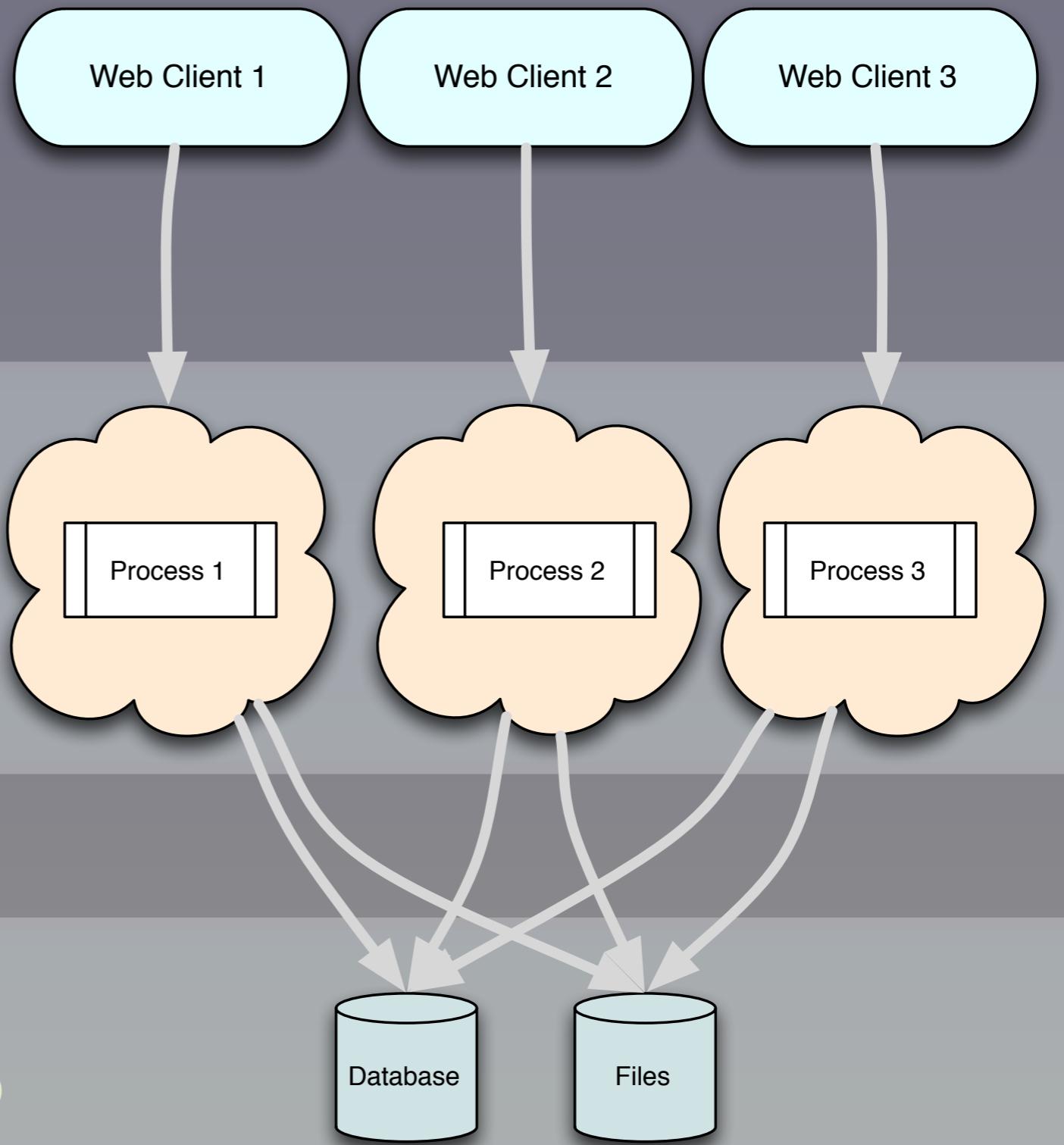
In a broader view, object models tend to push us towards centralized, complex systems that don't decompose well and stifle reuse and optimal deployment scenarios. FP code makes it easier to write smaller, focused services that we compose and deploy as appropriate. Each "ProcessN" could be a parallel copy of another process, for horizontal, "shared-nothing" scalability, or some of these processes could be other services...

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

- Data Size ↑
- Formal Schema ↓
- Data-Driven Programs ↑



- MapReduce

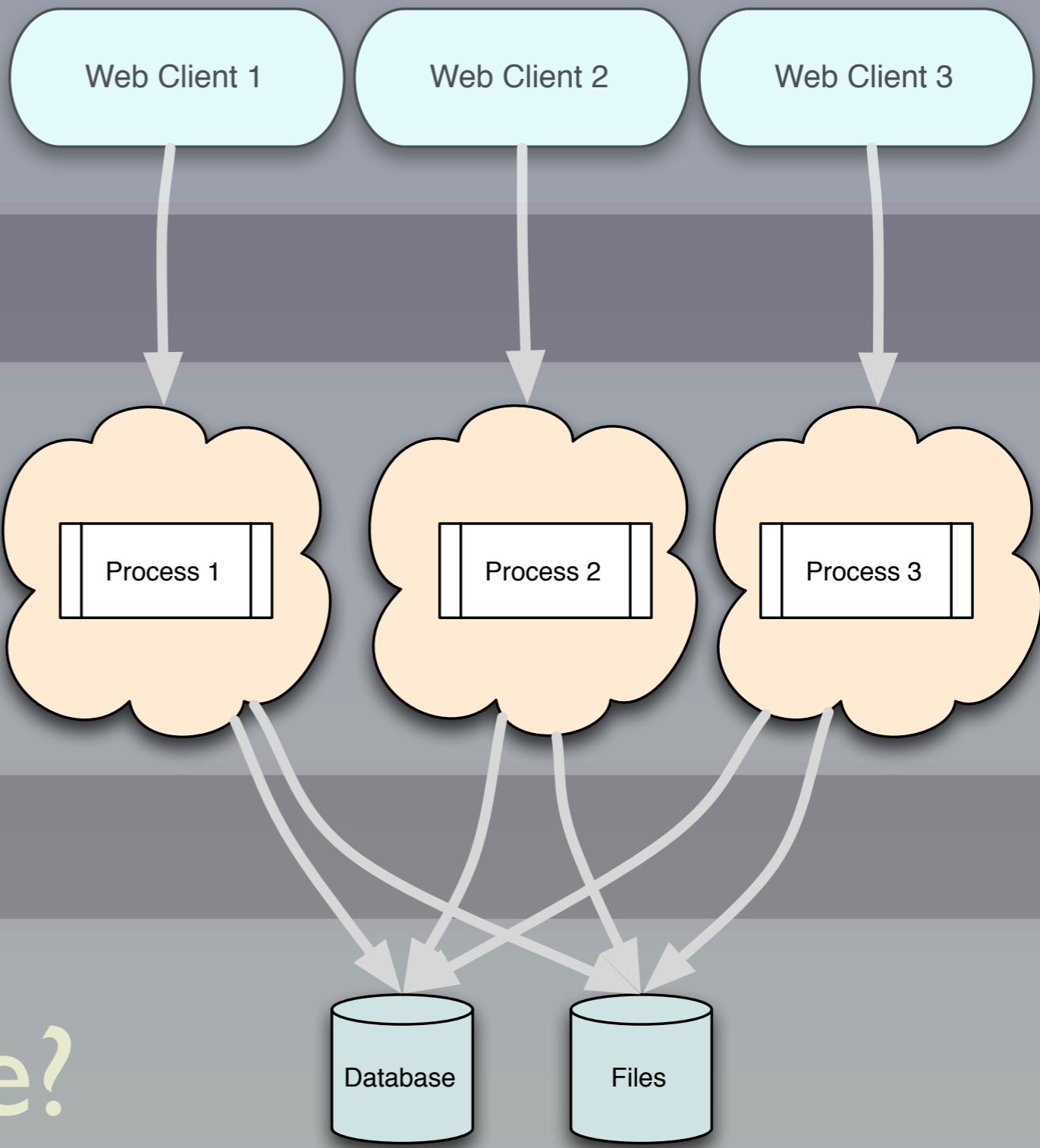


- Distributed FS

- JSON

- Node.js?

- JSON database?



# What Is MapReduce?



Tuesday, April 23, 13

“The Bean” on a sunny day – with some of my relatives ;)

Right © 2011

# MapReduce in Hadoop

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

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

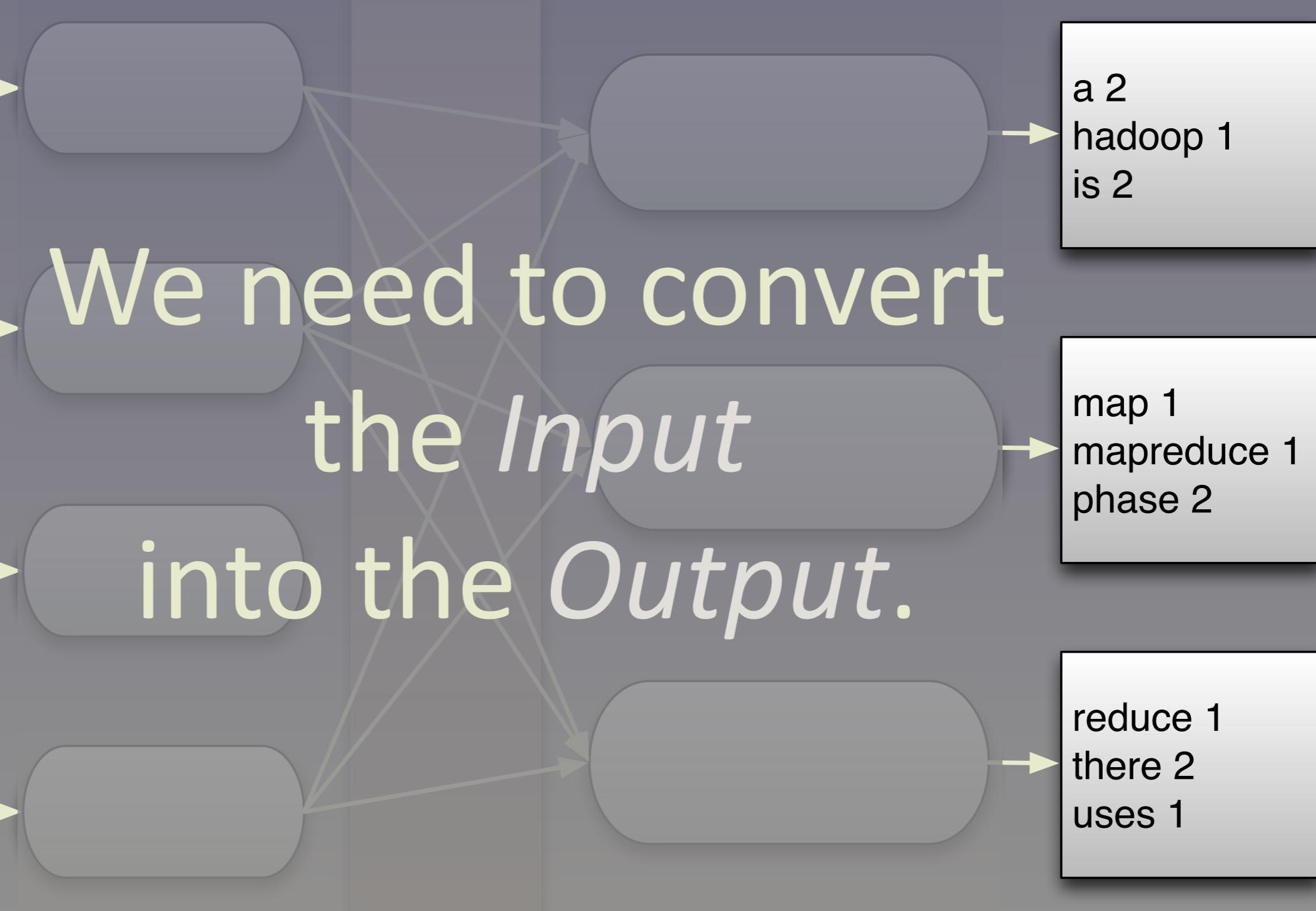
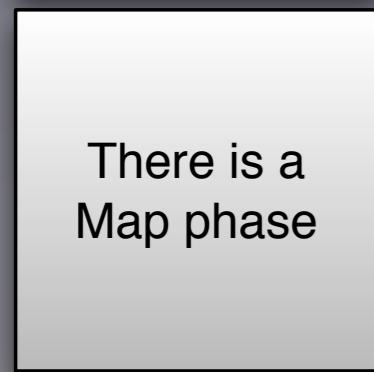
Input

Mappers

Sort,  
Shuffle

Reducers

Output



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.

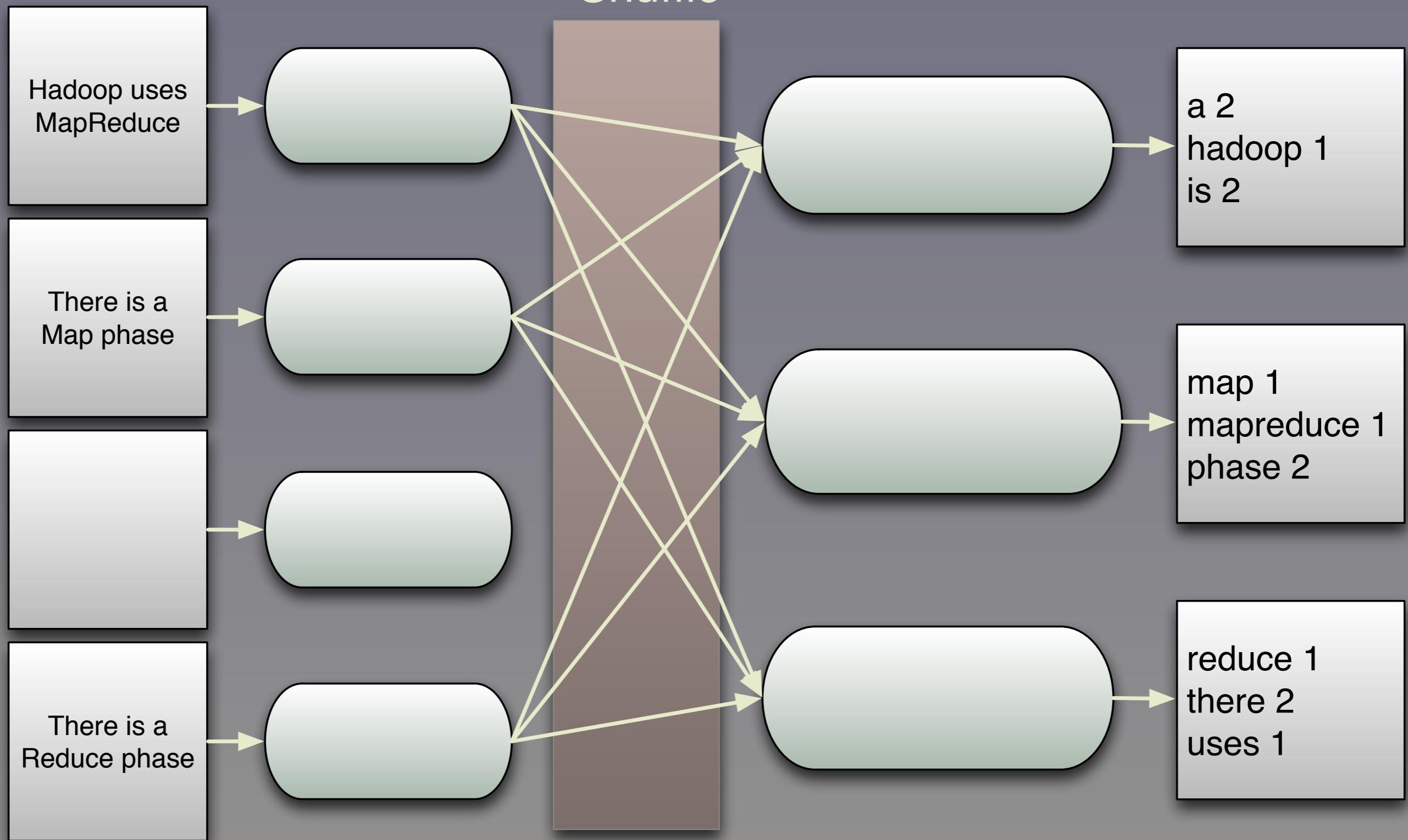
## Input

## Mappers

## Sort, Shuffle

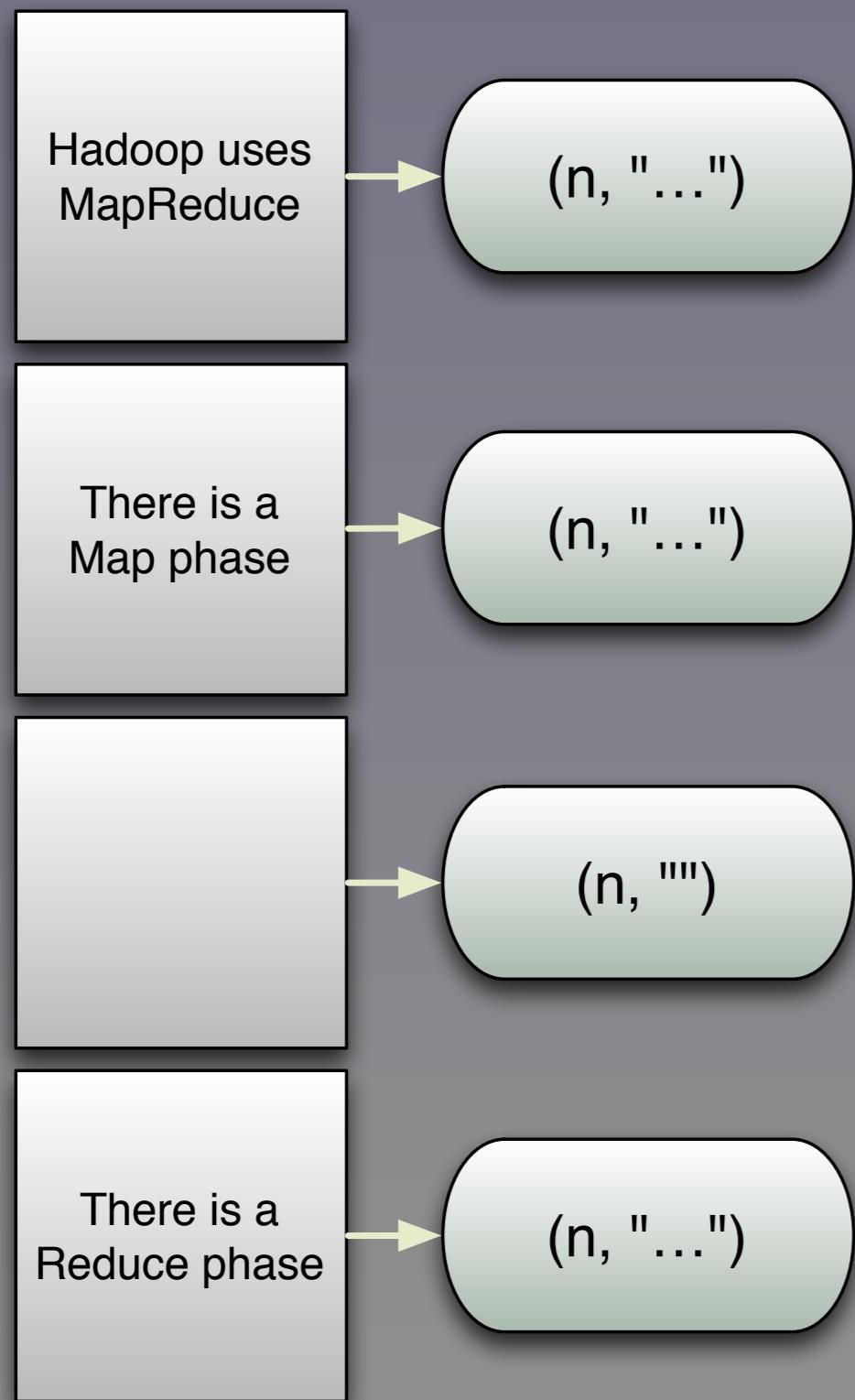
## Reducers

## Output



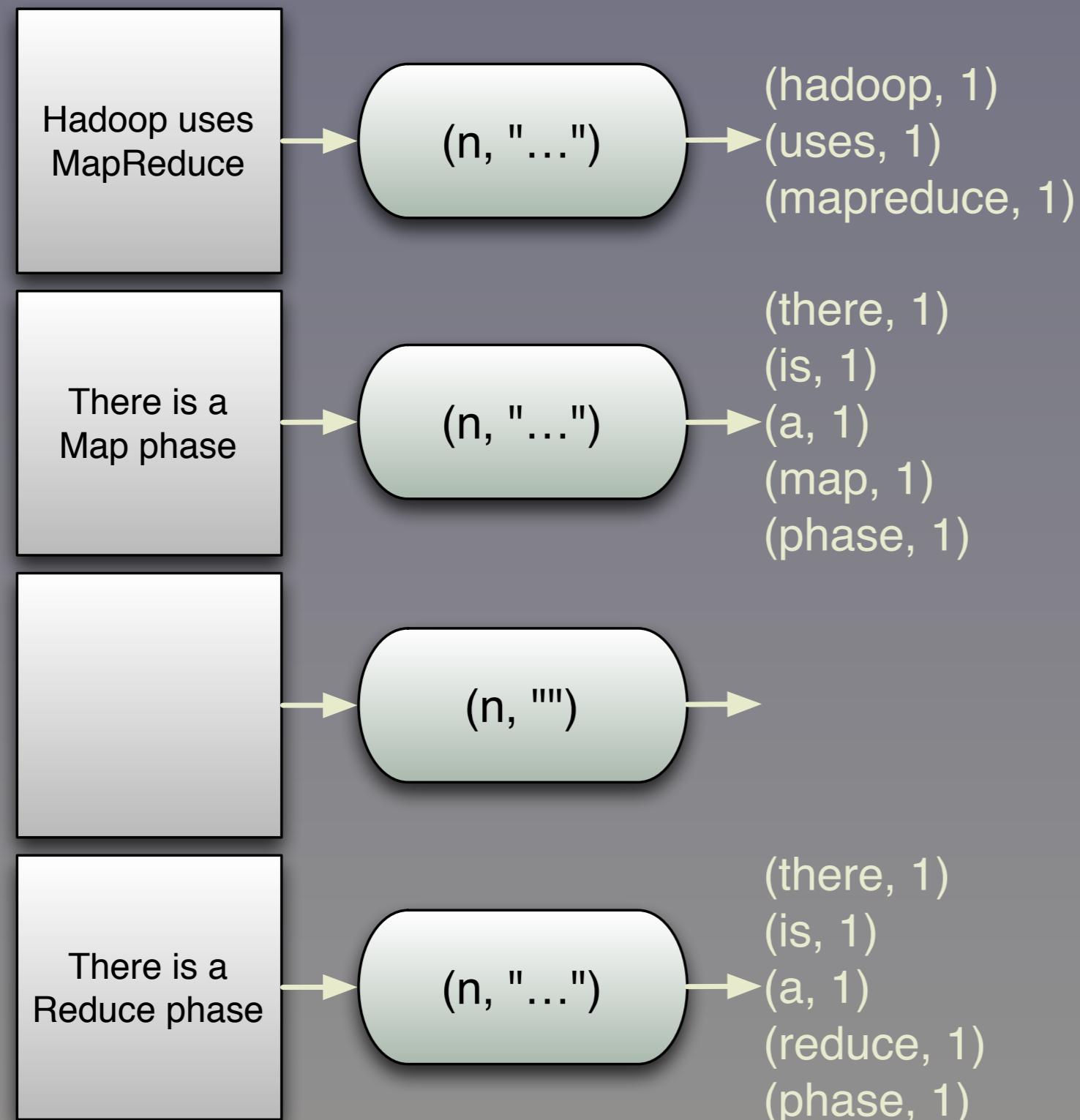
Here is a schematic view of the steps in Hadoop MapReduce. Each Input file is read by a single Mapper process (default: can be many-to-many, as we'll see later). The Mappers emit key-value pairs that will be sorted, then partitioned and "shuffled" to the reducers, where each Reducer will get all instances 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).

# Input Mappers



Each document gets a mapper. All data is organized into key-value pairs; each line will be a value and the offset position into the file will be the key, which we don't care about. I'm showing each document's contents in a box and 1 mapper task (JVM process) per document. Large documents might get split to several mapper tasks.  
The mappers tokenize each line, one at a time, converting all words to lower case and counting them...

# Input Mappers



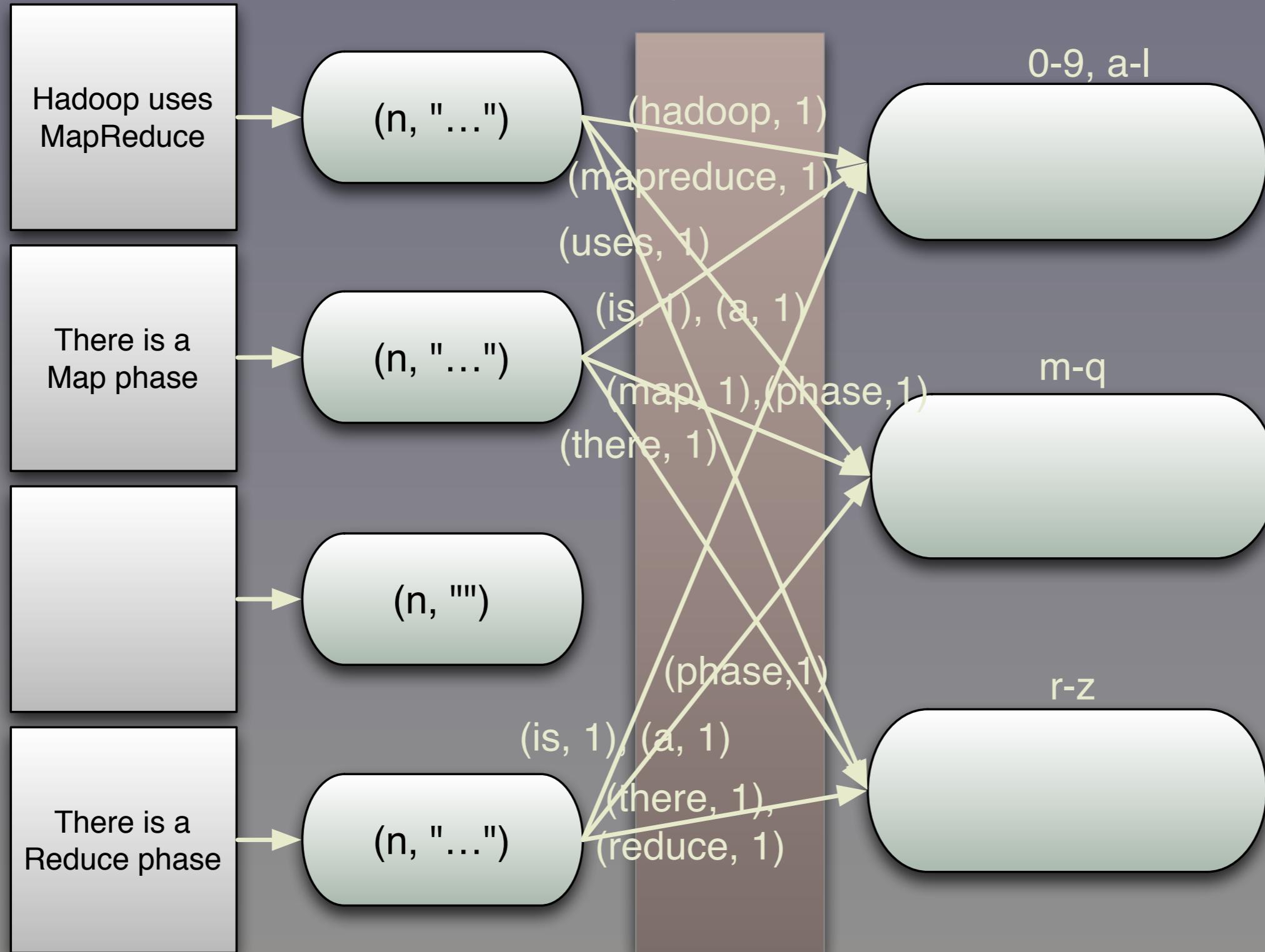
The mappers emit key-value pairs, where each key is one of the words, and the value is the count. In the most naive (but also most memory efficient) implementation, each mapper simply emits (word, 1) each time “word” is seen. However, this is IO inefficient! Note that the mapper for the empty doc. emits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

# Input

# Mappers

# Sort, Shuffle

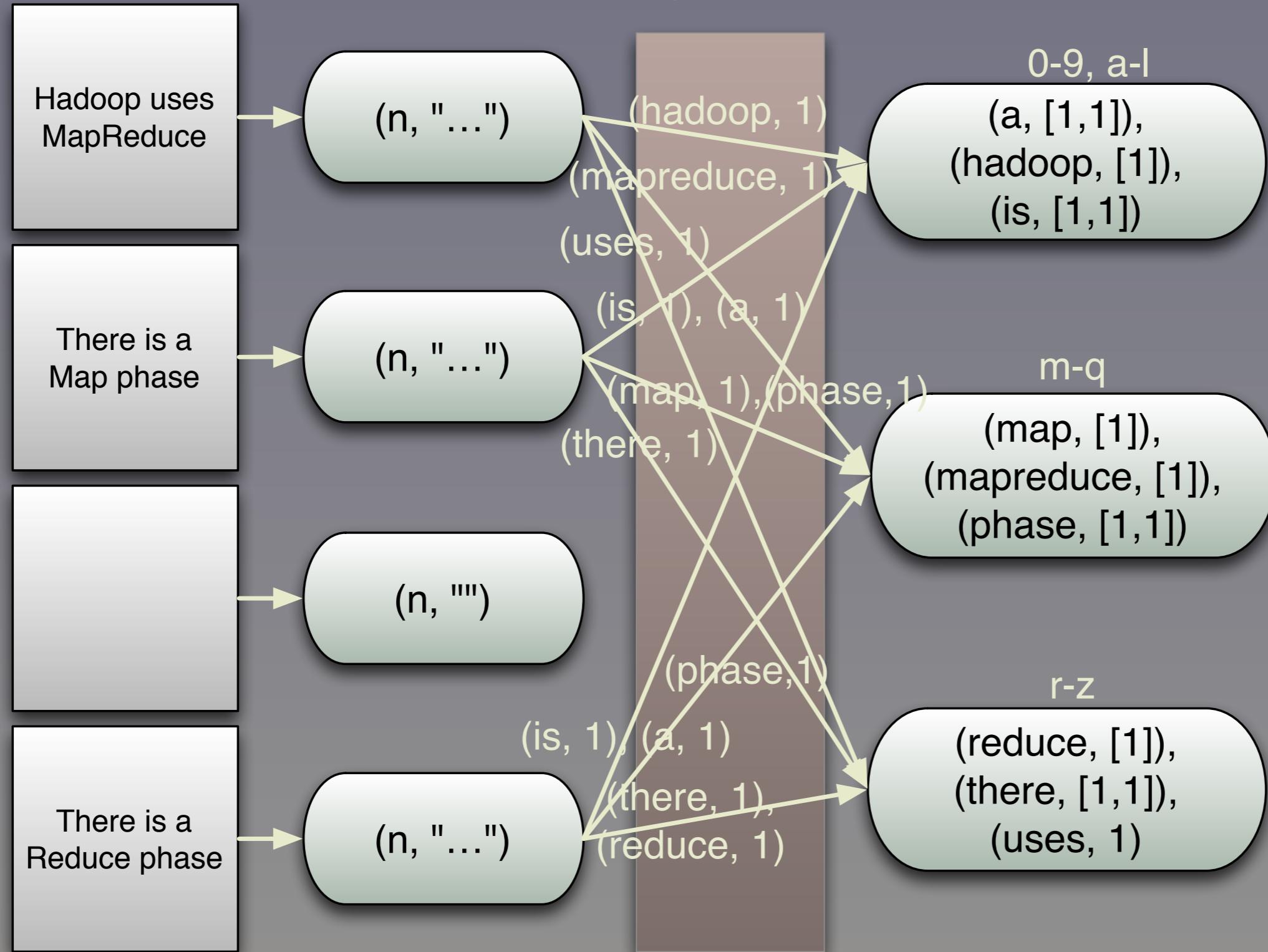
# Reducers



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.

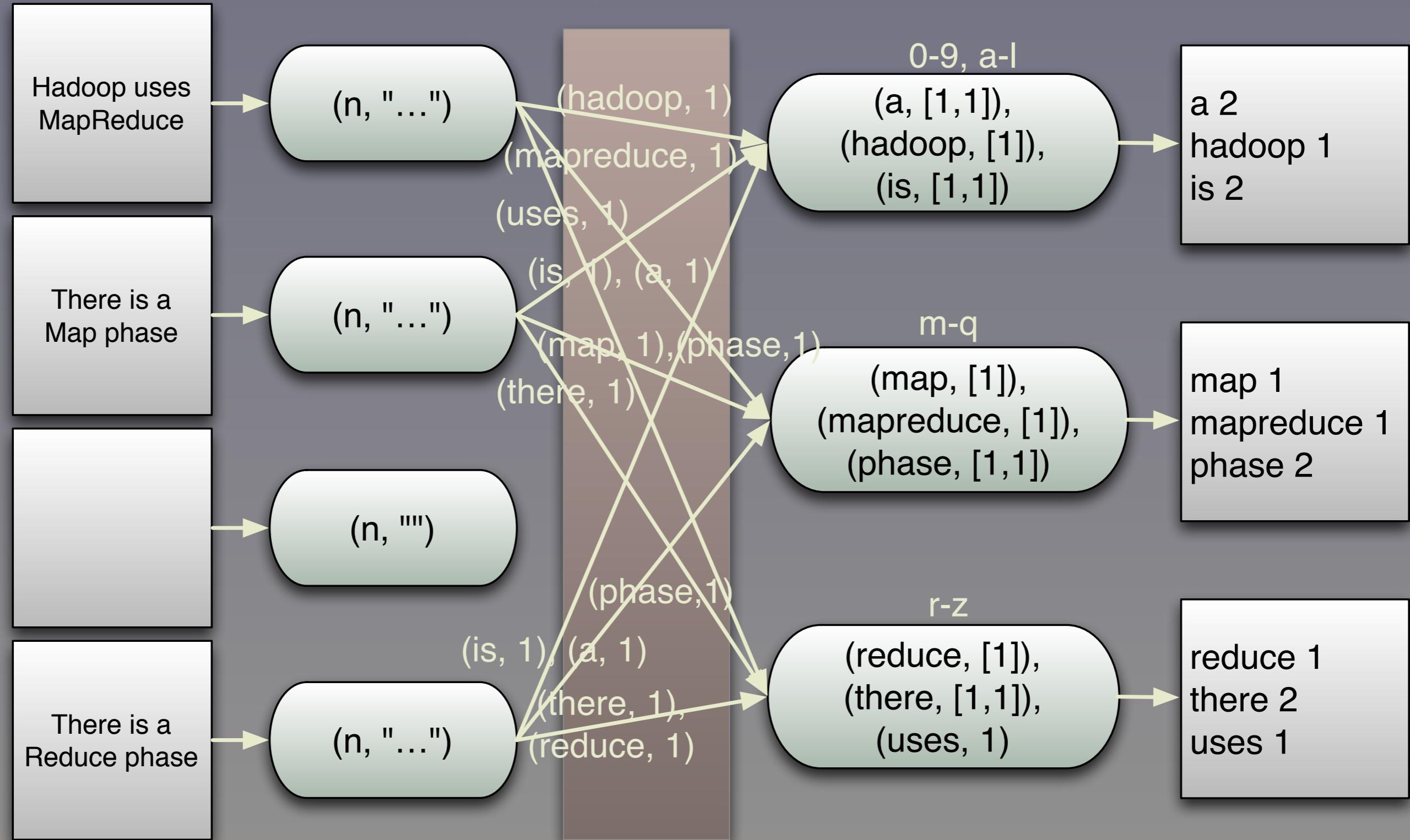
Note how we partitioned the reducers, by first letter of the keys. (By default, MR just hashes the keys and distributes them modulo # of reducers.)

# Input      Mappers      Sort, Shuffle      Reducers

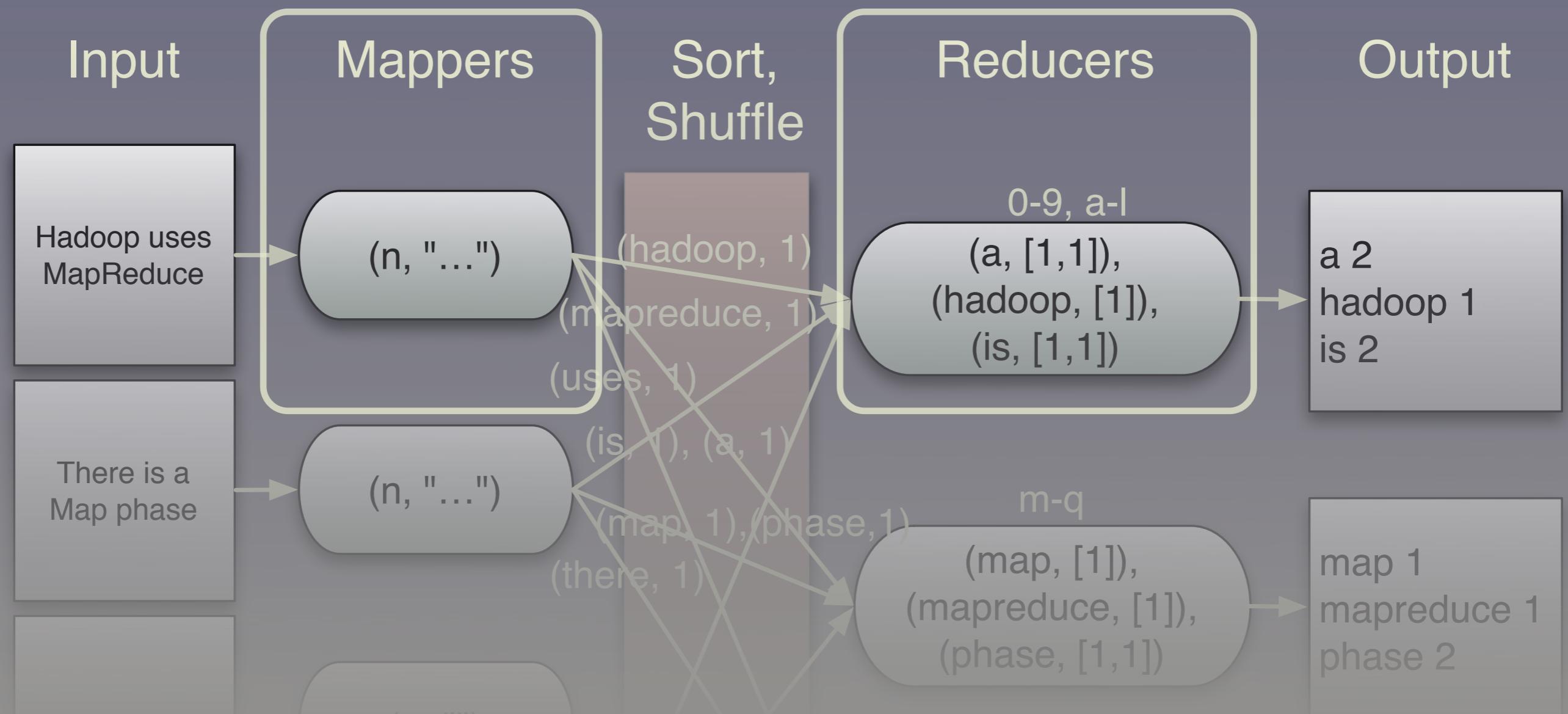


The reducers are passed each key (word) and a collection of all the values for that key (the individual counts emitted by the mapper tasks). The MR framework creates these collections for us.

# Input      Mappers      Sort, Shuffle      Reducers      Output



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.



## 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

Arguably, Hadoop is our  
best, general-purpose  
tool for scaling Big Data  
horizontally  
(at least today).

By design, Hadoop is  
*great for batch mode*  
data crunching.

*Not so great for event-  
stream processing.*

# MapReduce and Its Discontents

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

Is MapReduce the end of the story? Does it meet all our needs? Let's look at a few problems...  
Photo: Gratuitous Romantic beach scene, Ohio St. Beach, Feb. 2011.

# #1

It's hard to *implement*  
many *Algorithms*  
in *MapReduce*.

Even word count is not “obvious”. When you get to fancier stuff like joins, group-bys, etc., the mapping from the algorithm to the implementation is not trivial at all. In fact, implementing algorithms in MR is now a specialized body of knowledge...

MapReduce is very  
course-grained.

1-Map, 1-Reduce  
phase...

#2

For *Hadoop* in  
particularly,  
the *Java API* is  
*hard to use.*

```

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import java.util.StringTokenizer;

class WCMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

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

    @Override
    public void map(LongWritable key, Text valueDocContents,
                    OutputCollector<Text, IntWritable> output, Reporter reporter) {
        String[] tokens = valueDocContents.toString().split("\\s+");
        for (String wordString: tokens) {
            if (wordString.length > 0) {
                word.set(wordString.toLowerCase());
                output.collect(word, one);
            }
        }
    }
}

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

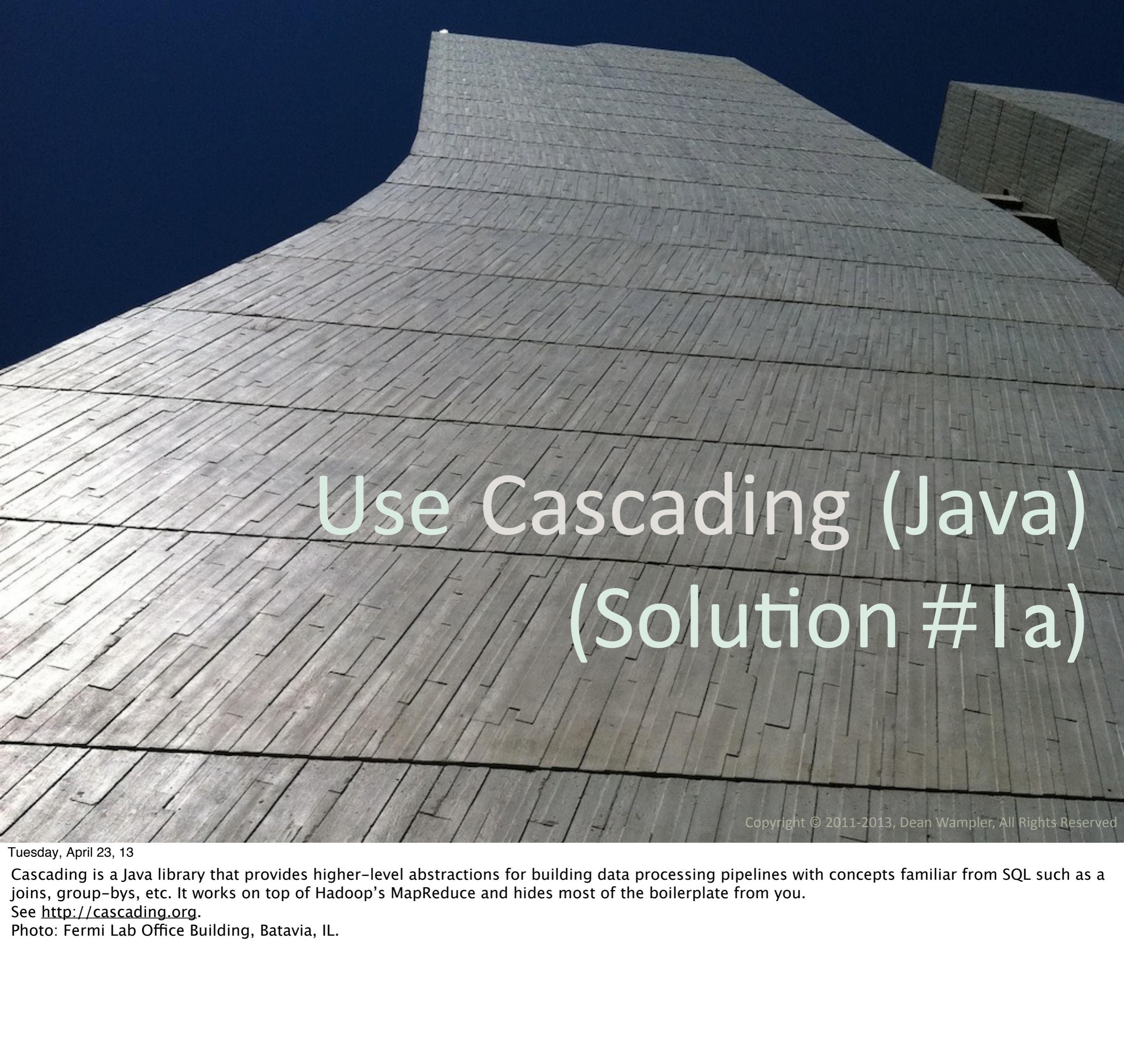
    public void reduce(Text keyword, java.util.Iterator<IntWritable> valuesCounts,
                      OutputCollector<Text, IntWritable> output, Reporter reporter) {
        int totalCount = 0;
        while (valuesCounts.hasNext()) {
            totalCount += valuesCounts.next().get();
        }
        output.collect(keyword, new IntWritable(totalCount));
    }
}

```

Tuesday, April 23, 13

This is intentionally too small to read and we're not showing the main routine, which doubles the code size. The algorithm is simple, but the framework is in your face. In the next several slides, notice which colors dominate. In this slide, it's green for types (classes), with relatively few yellow functions that implement actual operations.

The main routine I've omitted contains 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 ideas like relational-style joins and group-bys, the corresponding MapReduce code in this API gets complex and tedious very fast!



# Use Cascading (Java) (Solution #1a)

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

Cascading is a Java library that provides higher-level abstractions for building data processing pipelines with concepts familiar from SQL such as a joins, group-bys, etc. It works on top of Hadoop's MapReduce and hides most of the boilerplate from you.

See <http://cascading.org>.

Photo: Fermi Lab Office Building, Batavia, IL.

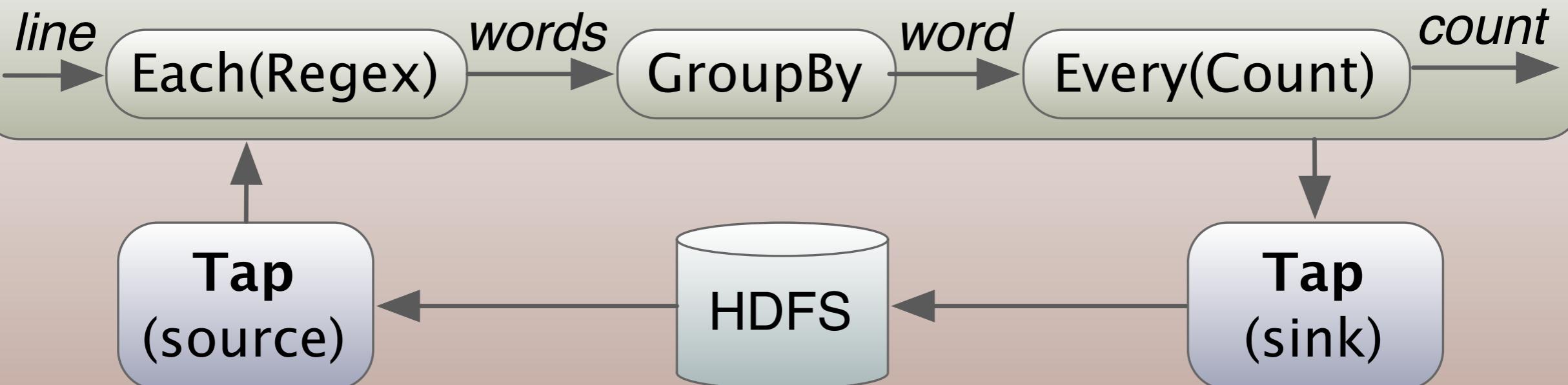
# Cascading Concepts

*Data flows consist of source and sink Taps connected by Pipes.*

# Word Count

## Flow

### Pipe ("word count assembly")



Schematically, here is what Word Count looks like in Cascading. See <http://docs.cascading.org/cascading/1.2/userguide/html/ch02.html> for details.

```

import org.cascading.*;
...
public class WordCount {
    public static void main(String[] args) {
        String inputPath = args[0];
        String outputPath = args[1];
        Properties properties = new Properties();
        FlowConnector.setApplicationJarClass( properties, Main.class );

        Scheme sourceScheme = new TextLine( new Fields( "line" ) );
        Scheme sinkScheme = new TextLine( new Fields( "word", "count" ) );
        Tap source = new Hfs( sourceScheme, inputPath );
        Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );

        Pipe assembly = new Pipe( "wordcount" );

        String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?=<\\pL)(?!\\pL)";
        Function function = new RegexGenerator( new Fields( "word" ), regex );
        assembly = new Each( assembly, new Fields( "line" ), function );
        assembly = new GroupBy( assembly, new Fields( "word" ) );
        Aggregator count = new Count( new Fields( "count" ) );
        assembly = new Every( assembly, count );

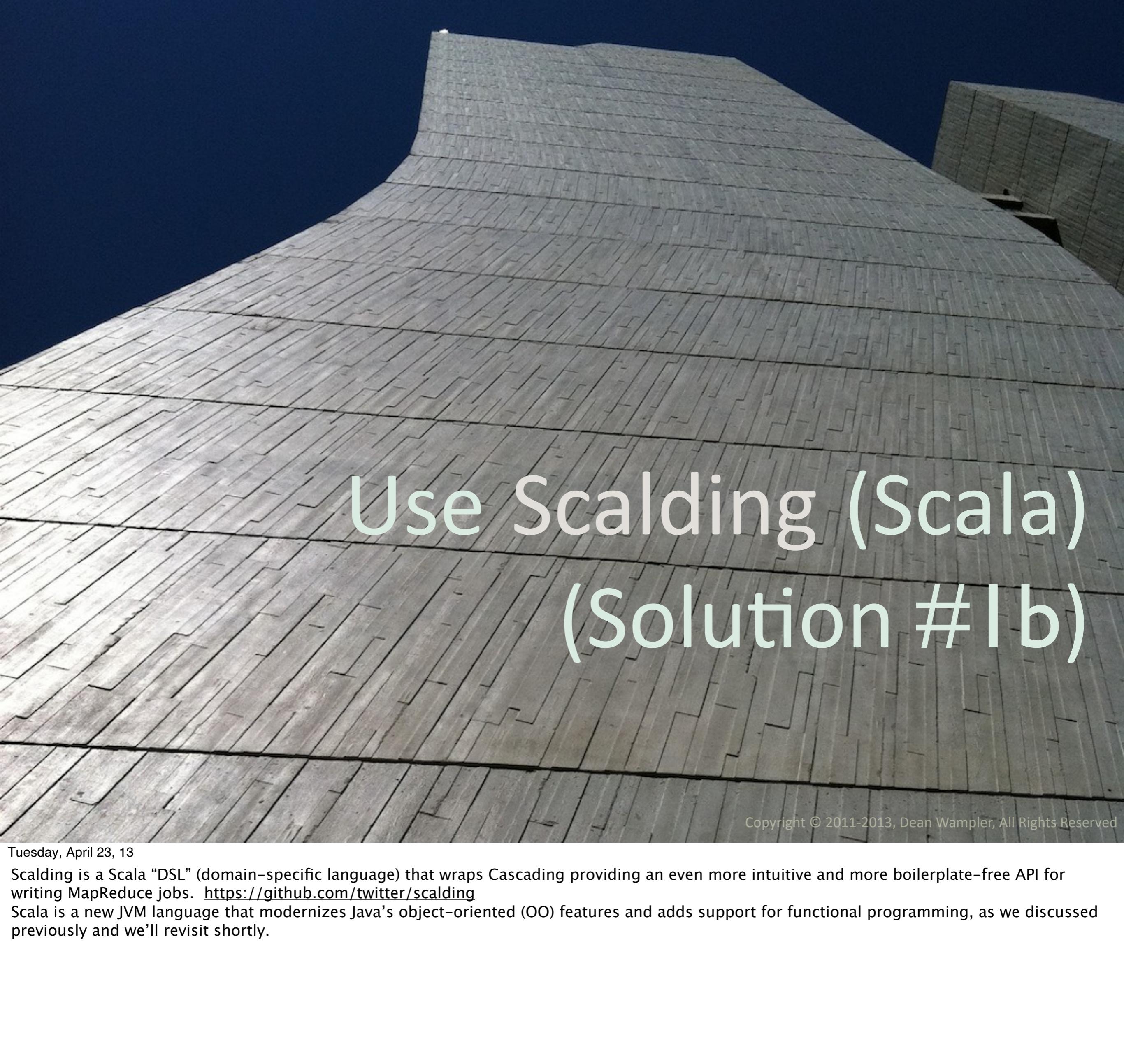
        FlowConnector flowConnector = new FlowConnector( properties );
        Flow flow = flowConnector.connect( "word-count", source, sink, assembly );
        flow.complete();
    }
}

```

Tuesday, April 23, 13

Here is the Cascading Java code. It's cleaner than the MapReduce API, because the code is more focused on the algorithm with less boilerplate, although it looks like it's not that much shorter. HOWEVER, this is all the code, where as previously I omitted the setup (main) code. See <http://docs.cascading.org/cascading/1.2/userguide/html/ch02.html> for details of the API features used here; we won't discuss them here, but just mention some highlights.

Note that there is still a lot of green for types, but at least the API emphasizes composing behaviors together.



# Use Scalding (Scala) (Solution #1b)

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

Scalding is a Scala “DSL” (domain-specific language) that wraps Cascading providing an even more intuitive and more boilerplate-free API for writing MapReduce jobs. <https://github.com/twitter/scalding>

Scala is a new JVM language that modernizes Java’s object-oriented (OO) features and adds support for functional programming, as we discussed previously and we’ll revisit shortly.

```
import com.twitter.scalding._

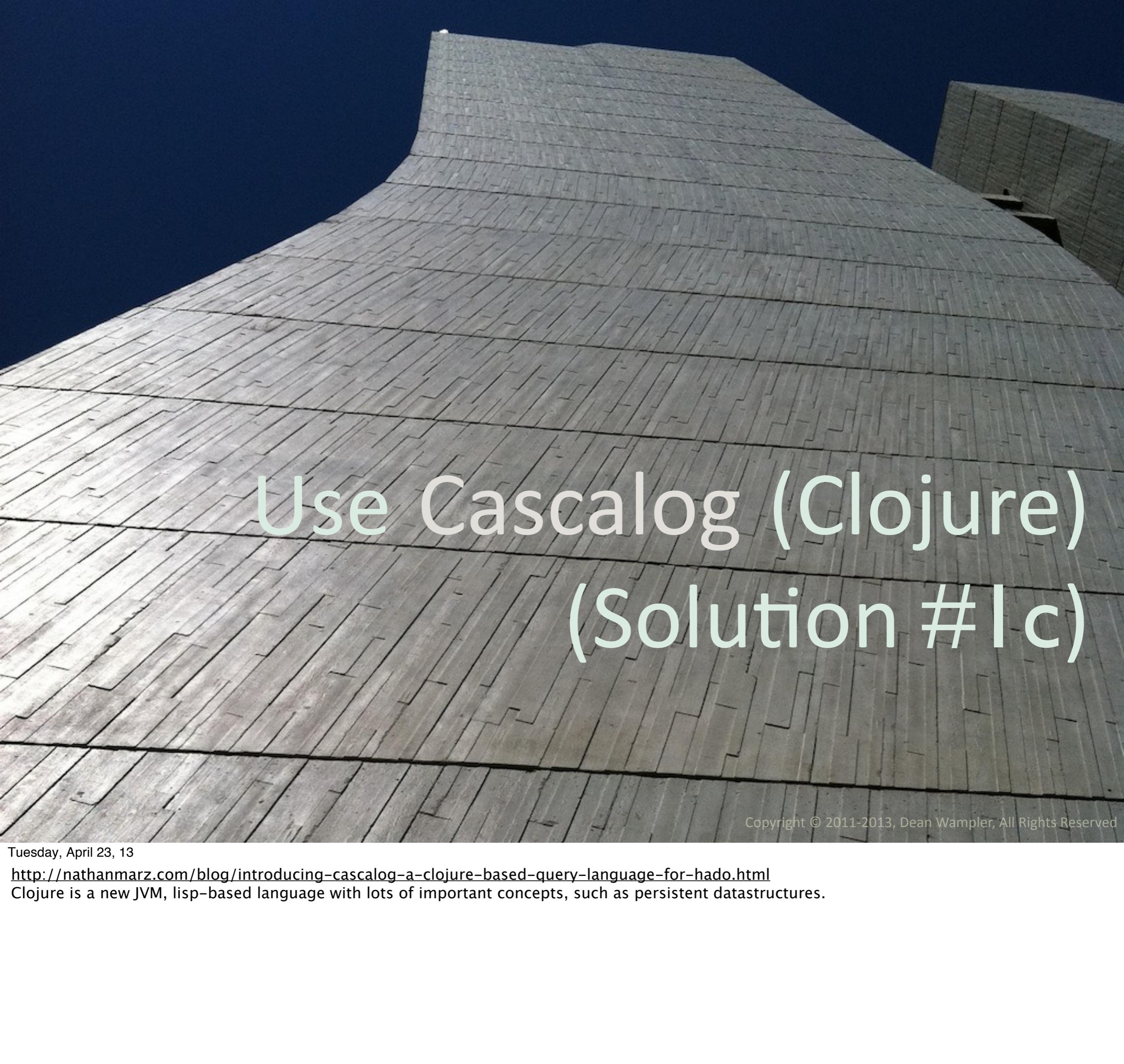
class WordCountJob(args: Args) extends Job(args) {
    TextLine( args("input") )
        .read
        .flatMap('line -> 'word) {
            line: String =>
            line.trim.toLowerCase
                .split("\\\\W+")
        }
        .groupBy('word) {
            group => group.size('count)
        }
    }
    .write(Tsv(args("output")))
}
```

That's It!!

This Scala code is almost pure domain logic with very little boilerplate. There are a few minor differences in the implementation. You don't explicitly specify the "Hfs" (Hadoop Distributed File System) taps. That's handled by Scalding implicitly when you run in "non-local" mode. Also, I'm using a simpler tokenization approach here, where I split on anything that isn't a "word character" [0-9a-zA-Z\_].

There is little green, in part because Scala infers type in many cases. There is a lot more yellow for the functions that do real work!

What if MapReduce, and hence Cascading and Scalding, went obsolete tomorrow? This code is so short, I wouldn't care about throwing it away! I invested little time writing it, testing it, etc.



# Use Cascalog (Clojure) (Solution #1c)

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

<http://nathanmarz.com/blog/introducing-cascalog-a-clojure-based-query-language-for-hadoop.html>

Clojure is a new JVM, lisp-based language with lots of important concepts, such as persistent datastructures.

```
(defn lowercase [w] (.toLowerCase w))  
  
(?-< (stdout) [?word ?count]  
      (sentence ?s)  
      (split ?s :> ?word1)  
      (lowercase ?word1 :> ?word)  
      (c/count ?count))
```

## Datalog-style queries

# Other Improved APIs:

- Crunch (Java) & Scrunch (Scala)
- Scoobi (Scala)
- ...

See <https://github.com/cloudera/crunch>.

Others include Scoobi (<http://nicta.github.com/scoobi/>) and Spark, which we'll discuss next.



# Use Spark (Not MapReduce) (Solution #2)

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

<http://www.spark-project.org/>

Why isn't it more widely used? 1) lack of commercial support, 2) only recently emerged out of academia.

# Spark is a Hadoop MapReduce alternative:

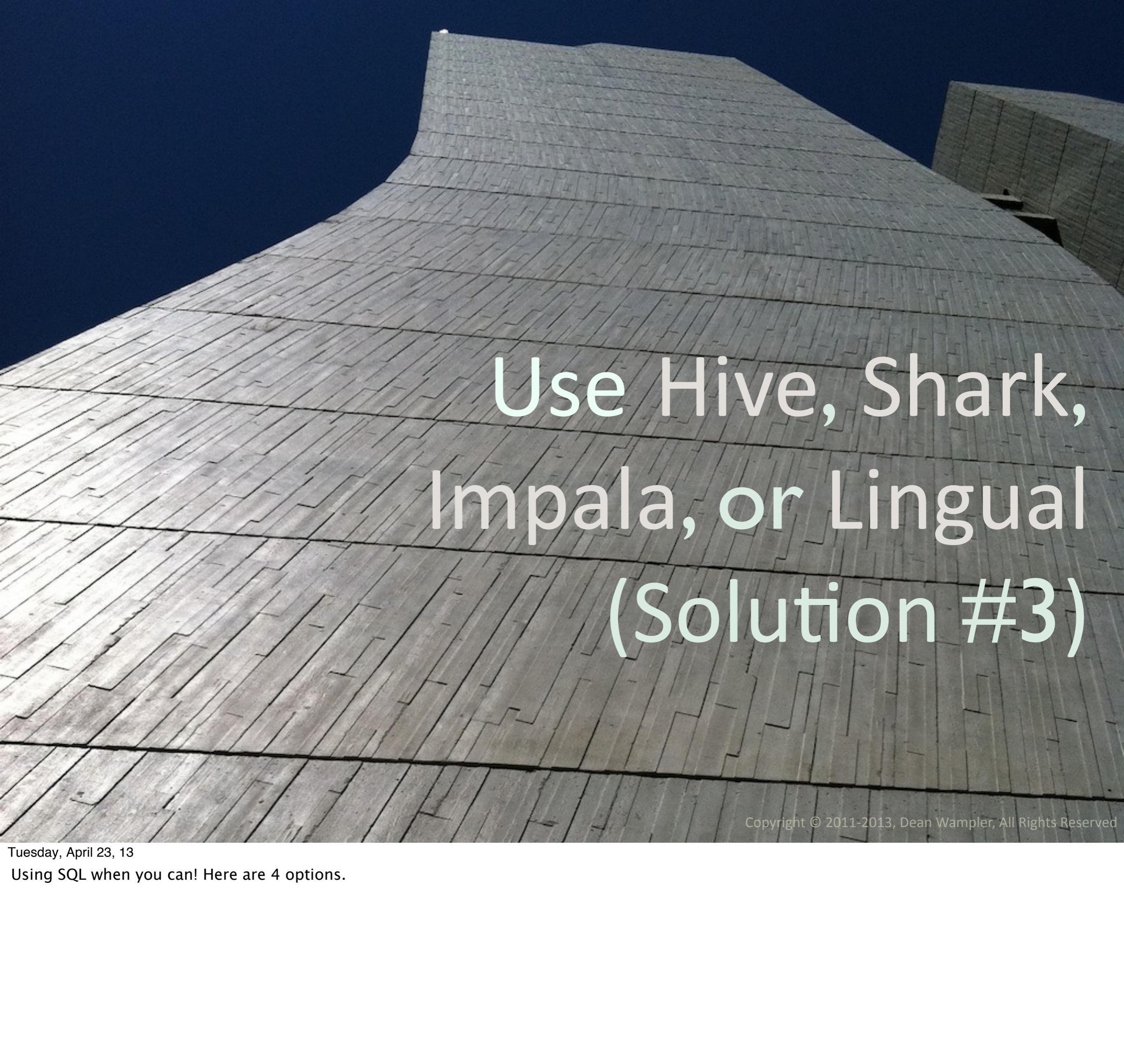
- Distributed computing with in-memory caching.
- Up to 30x faster than MapReduce.
- Developed by Berkeley AMP.

# Spark is a Hadoop MapReduce alternative:

- Originally designed for machine learning applications.

```
object WordCountSpark {  
    def main(args: Array[String]) {  
        val file = spark.textFile(args(0))  
        val counts = file.flatMap(  
            line => line.split("\\\\W+"))  
                .map(word => (word, 1))  
                .reduceByKey(_ + _)  
        counts.saveAsTextFile(args(1))  
    }  
}
```

Also small and concise!



# Use Hive, Shark, Impala, or Lingual (Solution #3)

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

Using SQL when you can! Here are 4 options.

# Use SQL when you can!

- Hive: SQL on top of MapReduce.
- Shark: Hive ported to Spark.
- Impala: HiveQL with new, faster back end.
- Lingual: SQL on Cascading.

See <http://hive.apache.org/> or my book for Hive, <http://shark.cs.berkeley.edu/> for shark, and <http://www.cloudera.com/content/cloudera/en/products/cloudera-enterprise-core/cloudera-enterprise-RTQ.html> for Impala. Impala is very new. It doesn't yet support all Hive features.

# Word Count in Hive SQL!

```
CREATE TABLE docs (line STRING);
LOAD DATA INPATH '/path/to/docs'
INTO TABLE docs;
```

```
CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
(SELECT explode(split(line, '\W+'))
 AS word FROM docs) w
GROUP BY word
ORDER BY word;
```

Works for Hive, Shark, and Impala

# Hive

- SQL dialect.
- Uses MapReduce back end.
  - So annoying latency.
- First SQL on Hadoop.
- Developed by Facebook.

# Shark

- HiveQL front end.
- Spark back end.
- Provides better performance.
- Developed by Berkeley AMP.

See <http://www.cloudera.com/content/cloudera/en/products/cloudera-enterprise-core/cloudera-enterprise-RTQ.html>. However, this was just announced a few ago (at the time of this writing), so it's not production ready quite yet...

# Impala

- HiveQL front end.
- C++ and Java back end.
- Provides up to 100x performance improvement!
- Developed by Cloudera.

See <http://www.cloudera.com/content/cloudera/en/products/cloudera-enterprise-core/cloudera-enterprise-RTQ.html>. However, this was just announced a few ago (at the time of this writing), so it's not production ready quite yet...

# Lingual

- ANSI SQL front end.
- Cascading back end.
- Same strengths/weaknesses for runtime performance as Hive.

# #3

It's not suitable for  
event processing  
("real-time").

For typical web/enterprise systems, "real-time" is up to 100s of milliseconds, so I'm using the term broadly (but following common practice in this industry). True real-time systems, such as avionics, have much tighter constraints.

# (Solution #4)



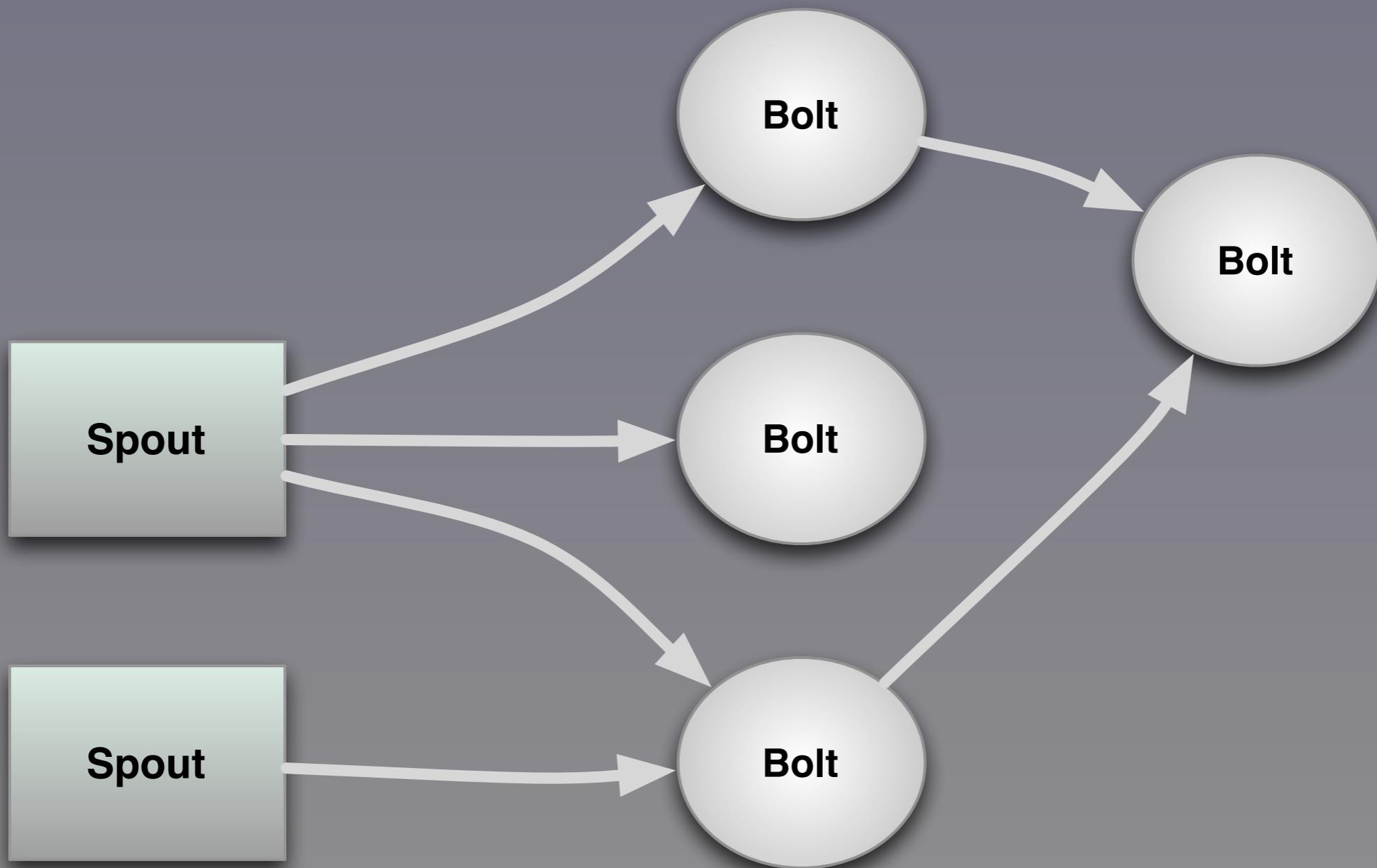
Storm!

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

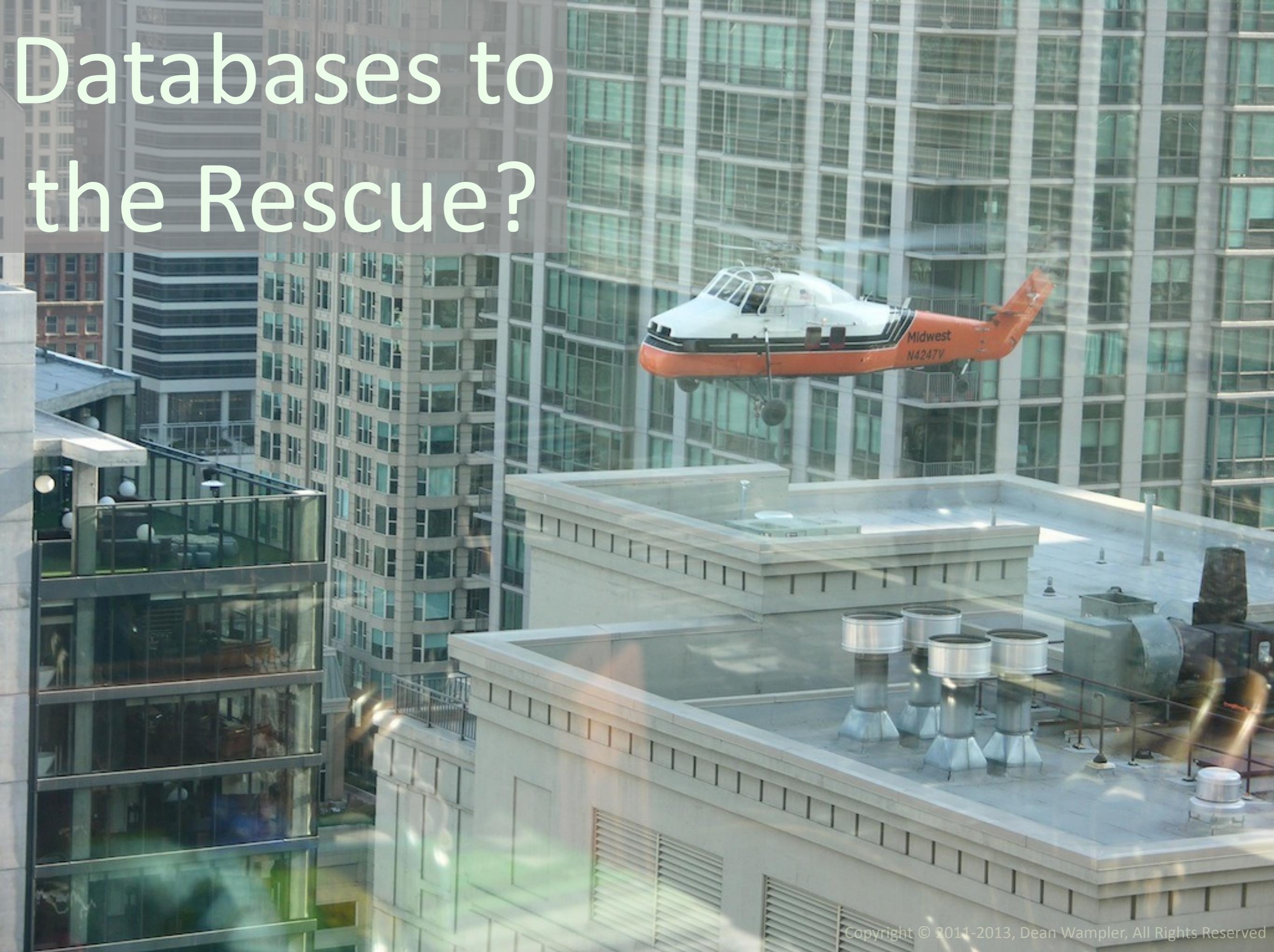
Photo: Top of the AON Building after a Storm passed through.

Storm implements  
reliable, distributed  
event processing.



In Storm terminology, Spouts are data sources and bolts are the event processors. There are facilities to support reliable message handling, various sources encapsulated in Spouts and various targets of output. Distributed processing is baked in from the start.

# Databases to the Rescue?



Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

# SQL or NoSQL Databases?

Databases are designed for fast, transactional updates.

So, consider a database for event processing.

# #4

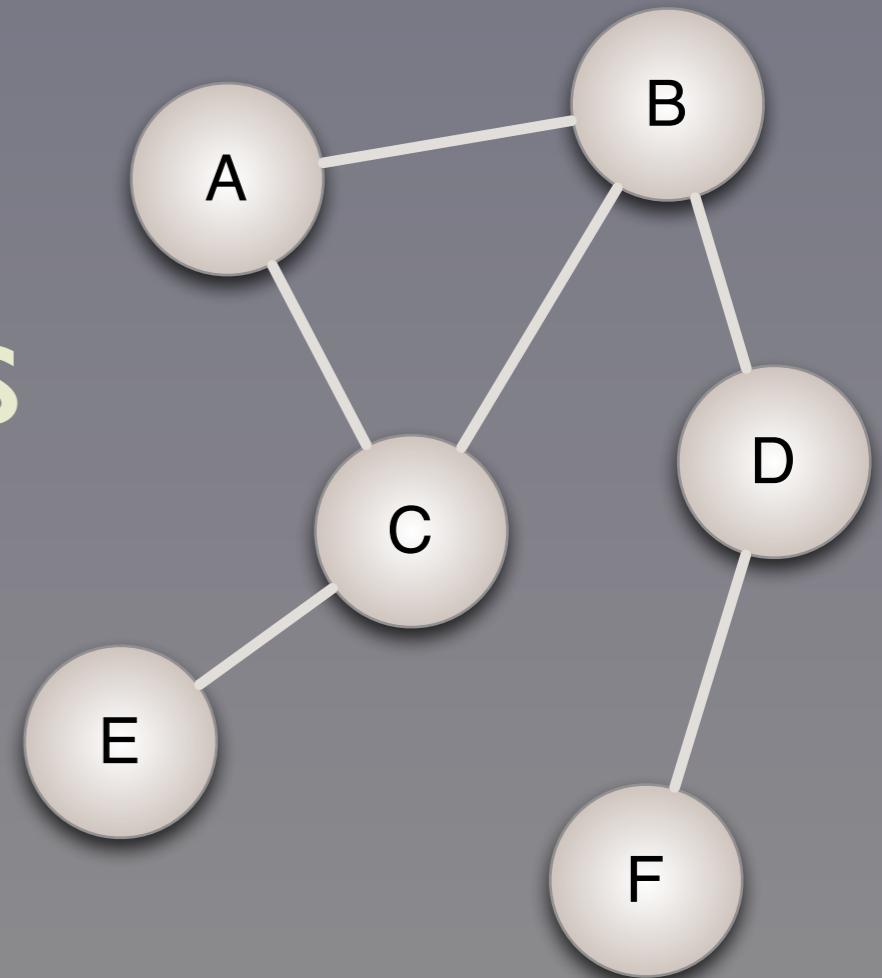
It's not ideal for  
*graph processing.*

# Google's Page Rank

Google invented MapReduce,  
... but MapReduce is not ideal for  
Page Rank and other graph  
algorithms.

# Why not MapReduce?

- 1 MR job for each iteration that updates all n nodes/edges.
- Graph saved to disk after each iteration.
- ...



# Use Graph Processing



## (Solution #5)

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

A good summary presentation: <http://www.slideshare.net/shatteredNirvana/pregel-a-system-for-largescale-graph-processing>  
Photo: Detail of the now-closed Esquire Movie Theater, a few blocks from here, Feb. 2011

# Google's Pregel

- Pregel: New graph framework for Page Rank.
- Bulk, Synchronous Parallel (BSP).
  - Graphs are first-class citizens.
  - Efficiently processes updates...

Pregel is the name of the river that runs through the city of Königsberg, Prussia (now called Kaliningrad, Ukraine). 7 bridges crossed the river in the city (including to 5 to 2 islands between river branches). Leonhard Euler invented graph theory when we analyzed the question of whether or not you can cross all 7 bridges without retracing your steps (you can't).

# Open-source Alternatives

- Apache Giraph.
- Apache Hama.
- Aurelius Titan.

All are  
somewhat  
immature.

# So, where are we??



Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

Now that we have cataloged some issues and solutions, let's recap and look forward.

Photo: Lake Michigan, Feb. 2011.



# Hadoop is the Enterprise Java Beans of our time.

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

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’s a 1st generation solution that mostly works okay and people do get work done with it, 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, specifically around Hadoop. The functional programming community, is well positioned to create it...

# MapReduce is waning

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

We've seen a lot of issues with MapReduce. Already, alternatives are being developed, either general options, like Spark and Storm, or special-purpose built replacements, like Impala. Let's consider other options...

# Successful replacements will be based on Functional Programming

```
import com.twitter.scalding._

class WordCountJob(args: Args) extends Job(args) {
  TextLine( args("input") )
    .read
    .flatMap('line -> 'word) {
      line: String =>
        line.trim.toLowerCase
        .split("\\\\w+")
    }
    .groupBy('word) {
      group => group.size('count) }
  }
  .write(Tsv(args("output")))
}
```

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

FP is such a natural fit for the problem that any attempts to build big data systems without it will be handicapped and probably fail.

Let's consider other MapReduce options...

Incidentally, concurrency  
has been called the killer  
app for FP.

Big Data is also  
a killer app, IMHO.

I think big data may drive FP adoption just as much as concurrency concerns, if not more so. Why? Because I suspect more developers will need to get “good” at data, vs. good at concurrency.

# Purpose-built Tools



Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

I see that a trend where completely generic tooling is giving way to more “purpose-built” tooling...

New Hadoop  
file formats, optimize  
access (e.g., Parquet).

New compute engines  
(e.g., Impala).

In the quest for ever better performance over massive datasets, the generic file formats in Hadoop and MapReduce are hitting a performance wall (although not everyone agrees). Parquet is column oriented & contains the data schema, like Thrift, Avro, and Protobuf. It will be exploited to optimize queries over massive data sets, much faster than the older file formats. Similarly, Impala is purpose built optimized query engine (that relies on Parquet).

Machine Learning and  
other advanced analytics  
are hard to write  
and perform poorly  
on Hadoop...

... but alternatives are emerging, including Spark, Storm, and proprietary solutions.



# SQL Strikes Back!

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Tuesday, April 23, 13

NoSQL solves lots of requirements better than traditional RDBMSs, but people loves them some SQL!!

# Hadoop owes a lot of its popularity to Hive!

Some “NoSQL”  
databases have or are  
adding query languages  
(e.g., Cassandra,  
MongoDB).

“NewSQL” databases  
are bringing NoSQL  
performance to the  
relational model.

# Examples

- Google Spanner and F1.
- NuoDB.
- VoltDB.

Spanner is the successor to BigTable. It is a globally-distributed database (consistency is maintained using the Paxos algo. and hardware synchronized clocks through GPS and atomic clocks!) Each table requires a primary key. F1 is an RDBMS built on top of it.

NuoDB is a cloud based RDBMS.

VoltDB is an example “in-memory” database, which are ideal for lots of small transactions that leverage indexing and rarely require full table scans.

# Two Final Emerging Trends...



Tuesday, April 23, 13

Both are examples of technologies focused on particular problems.  
photo: Trump hotel and residences on the Chicago River.

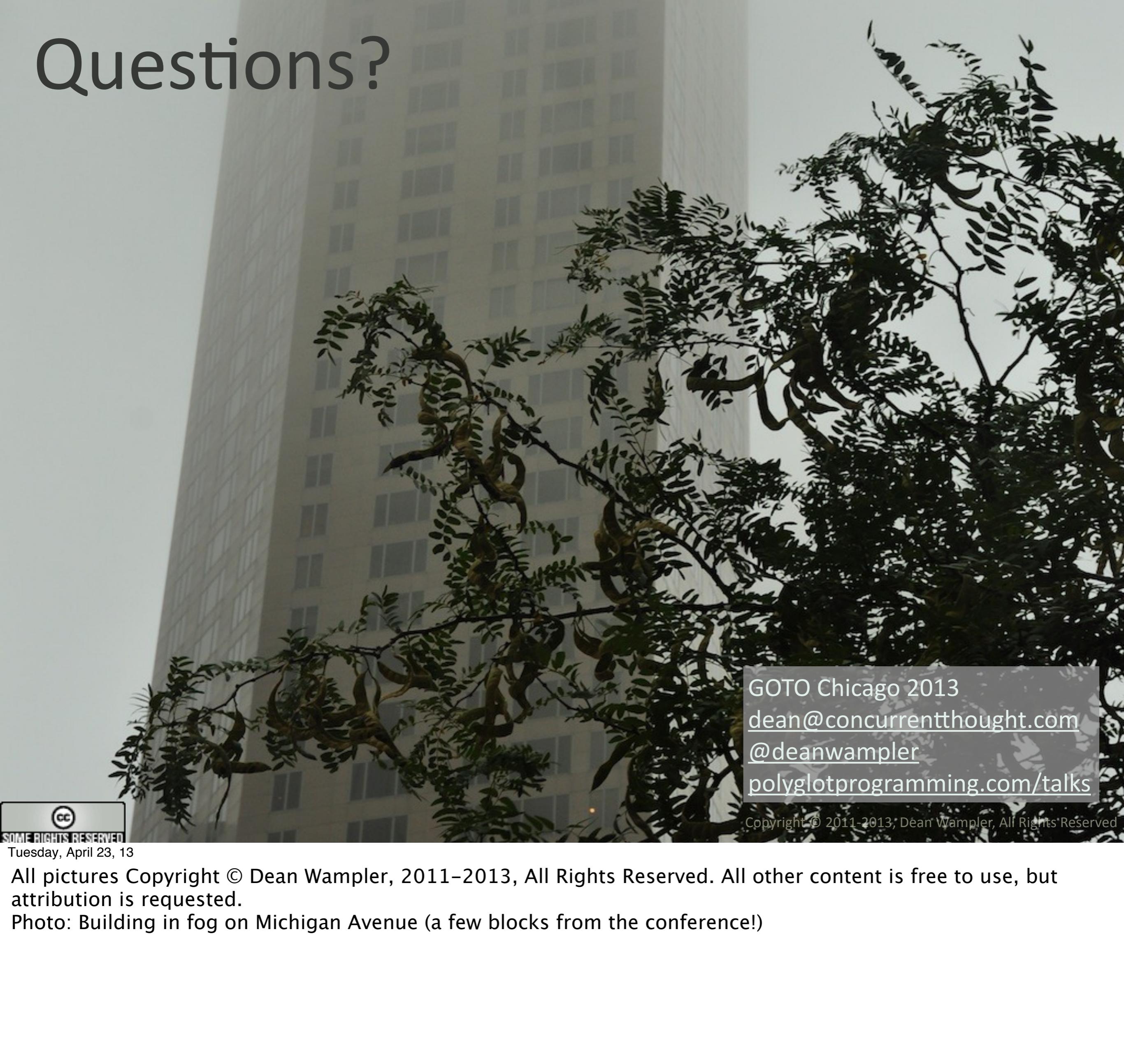
# Search

- Lucene and Solr.
- Elasticsearch.

# “Probabilistic Programming”

- Languages for Probabilistic Graphical Models??
- Bayesian Networks.
- Markov Chains.
- ...

# Questions?



GOTO Chicago 2013  
[dean@concurrentthought.com](mailto:dean@concurrentthought.com)  
[@deanwampler](https://twitter.com/deanwampler)  
[polyglotprogramming.com/talks](http://polyglotprogramming.com/talks)



Tuesday, April 23, 13

All pictures Copyright © Dean Wampler, 2011–2013, All Rights Reserved. All other content is free to use, but attribution is requested.

Photo: Building in fog on Michigan Avenue (a few blocks from the conference!)

Copyright © 2011-2013, Dean Wampler, All Rights Reserved