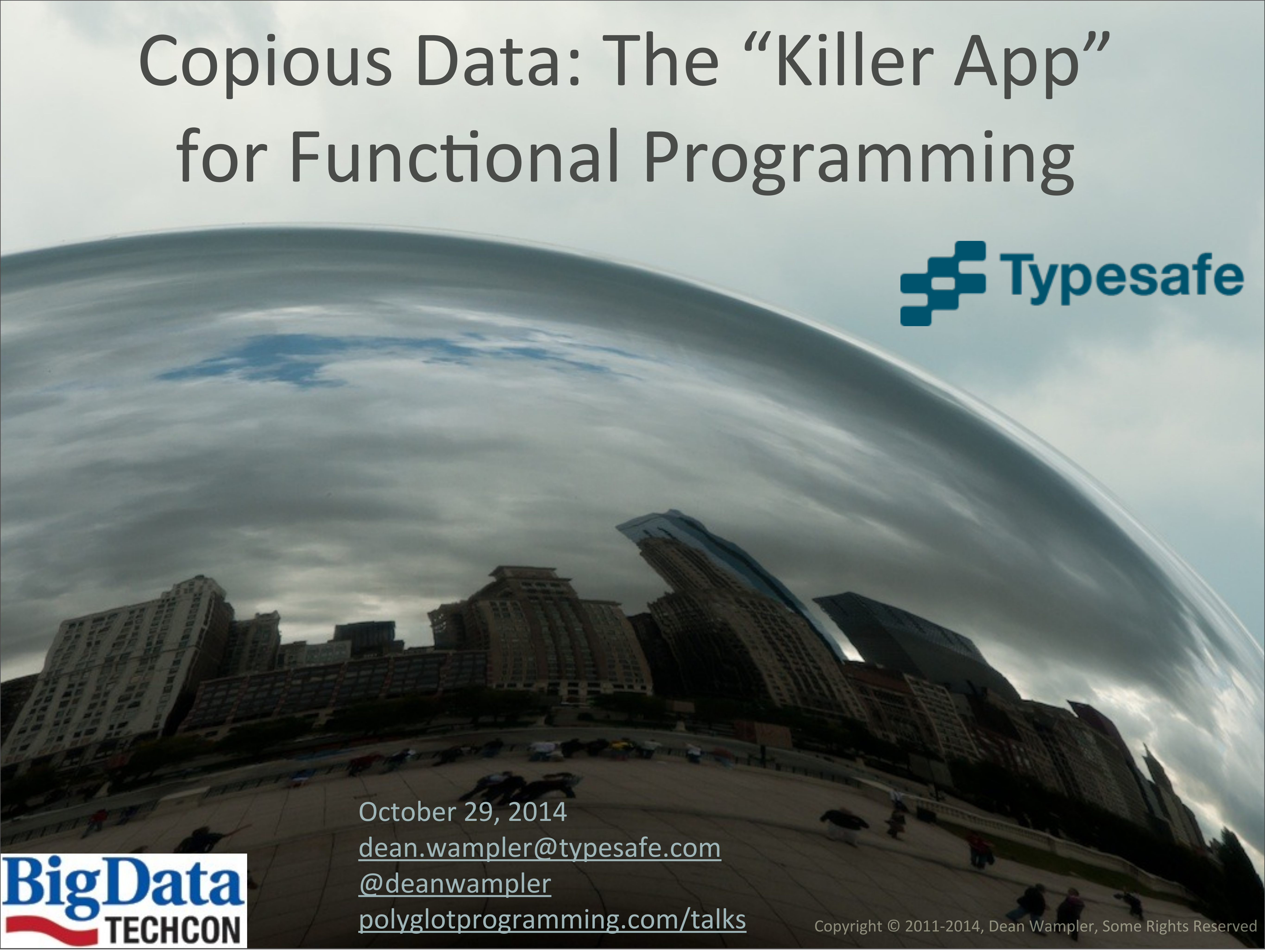


Copious Data: The "Killer App" for Functional Programming

October 29, 2014

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

Saturday, October 11, 14

Photo: Cloud Gate (a.k.a. "The Bean") in Millenium Park, Chicago, Illinois, USA

Architect for Big Data Products at Typesafe

Dean Wampler...

Saturday, October 11, 14

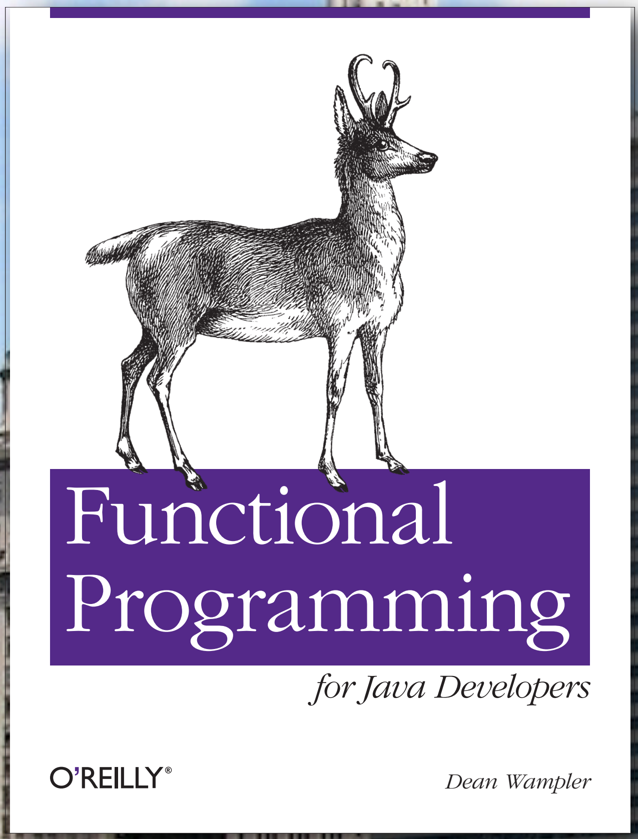
Typesafe builds tools for creating Reactive Applications, http://typesafe.com/platform, including Spark http://typesafe.com/reactive-big-data. See also the Reactive Manifesto, http://www.reactivemanifesto.org/

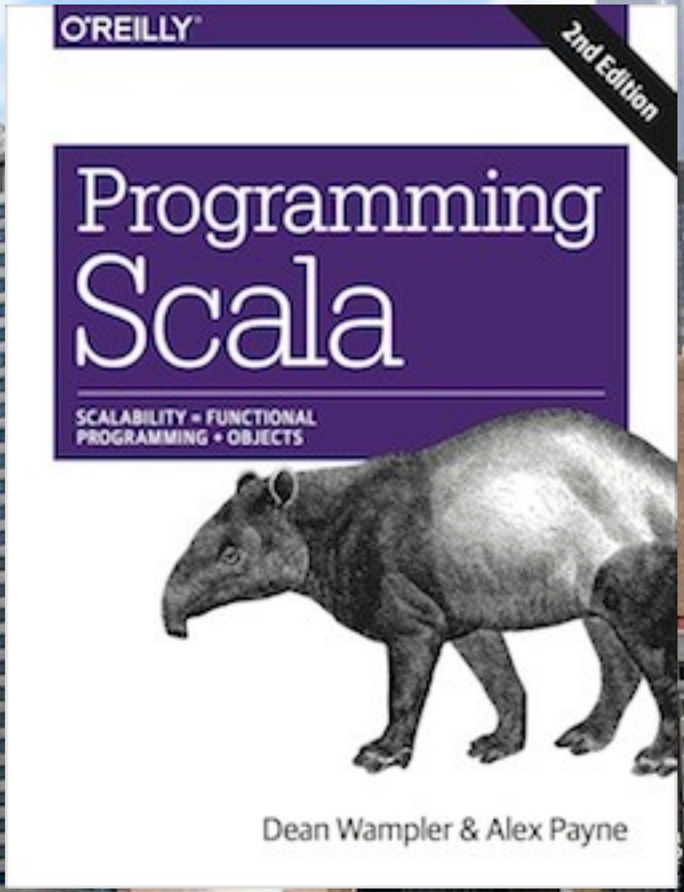
Photo: The Chicago River

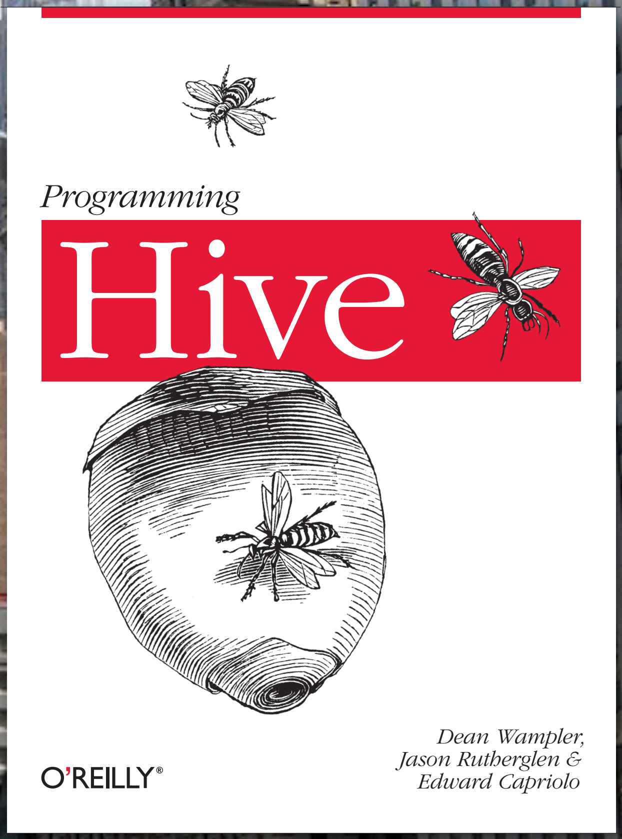
Founder,
Chicago-Area Scala Enthusiasts
and co-organizer,
Chicago Hadoop User Group

Dean Wampler...

I've been doing Scala for 6 years and Big Data for 3.5 years.




O'REILLY
2nd Edition
Programming Scala
SCALABILITY = FUNCTIONAL PROGRAMMING + OBJECTS
Dean Wampler & Alex Payne

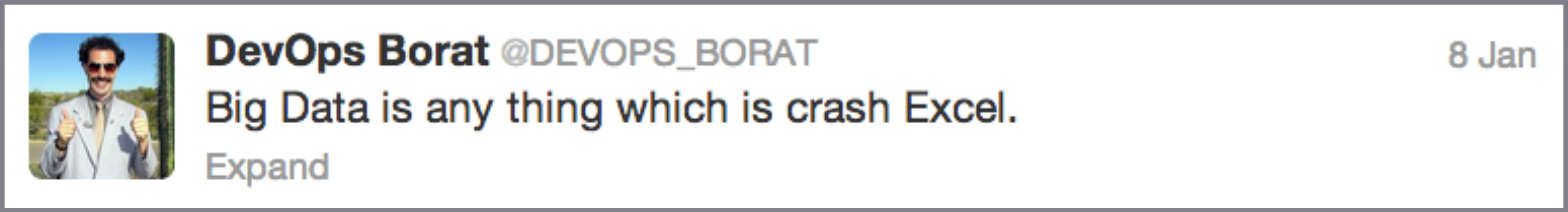


Dean Wampler...

Saturday, October 11, 14

My books...

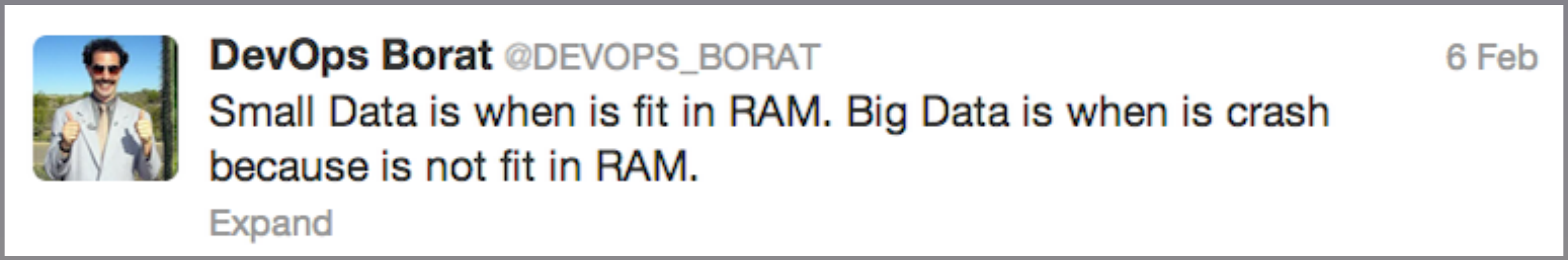
What Is Big … err… “Copious” Data?



DevOps Borat @DEVOPS_BORAT 8 Jan

Big Data is any thing which is crash Excel.

Expand



DevOps Borat @DEVOPS_BORAT 6 Feb

Small Data is when is fit in RAM. Big Data is when is crash because is not fit in RAM.

Expand

Copious Data

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

Hat tip: Bob Korbus

"Big Data" 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

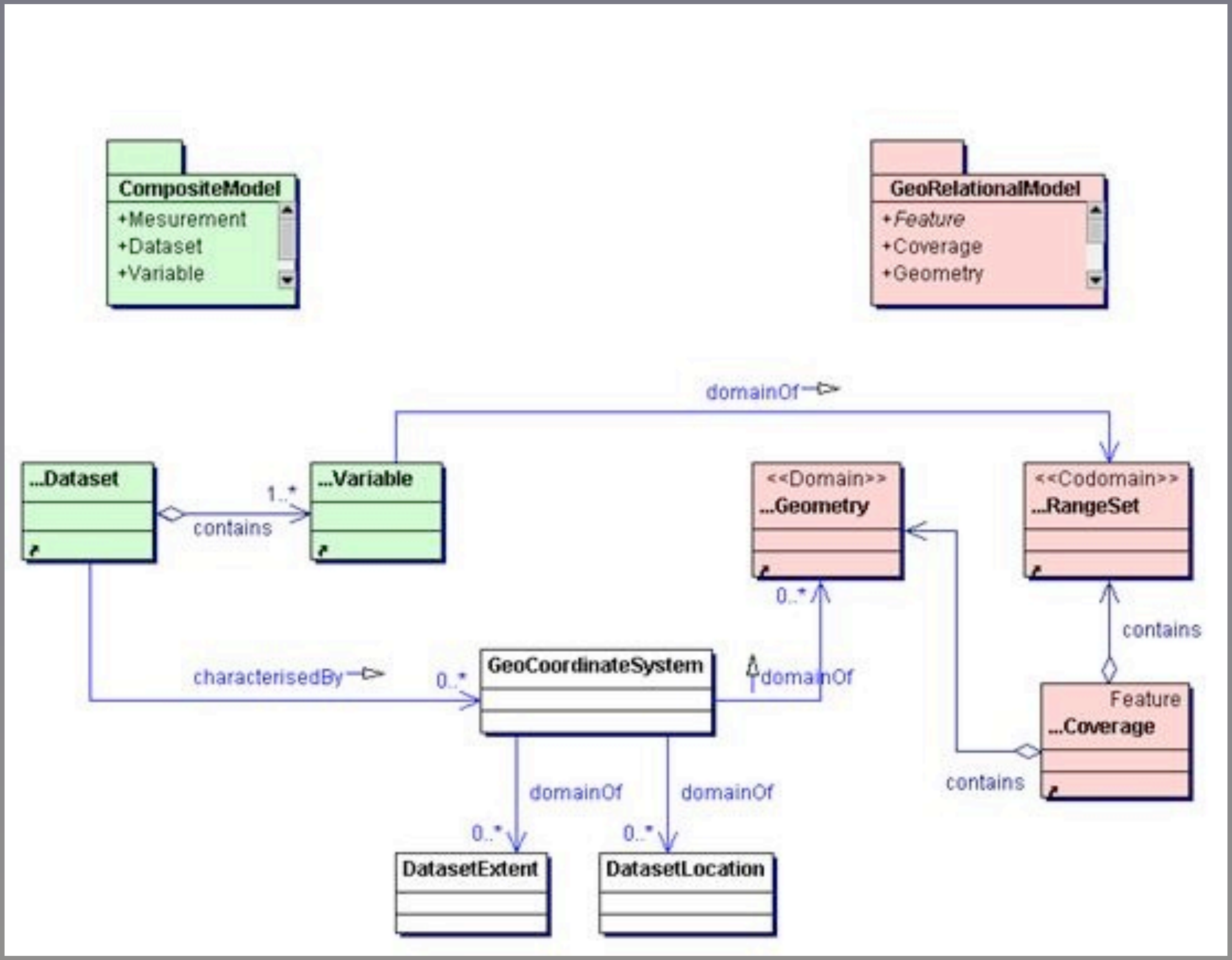
Three prevailing trends driving data-centric computing.
Photo: Prizker Pavilion, Millenium Park, Chicago (designed by Frank Gehry)

Data Size ⬆

Saturday, October 11, 14

Data volumes are obviously growing… rapidly.
Facebook now has over 600PB (Petabytes) of data in Hadoop clusters!

Formal Schemas ⬇



Saturday, October 11, 14

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...

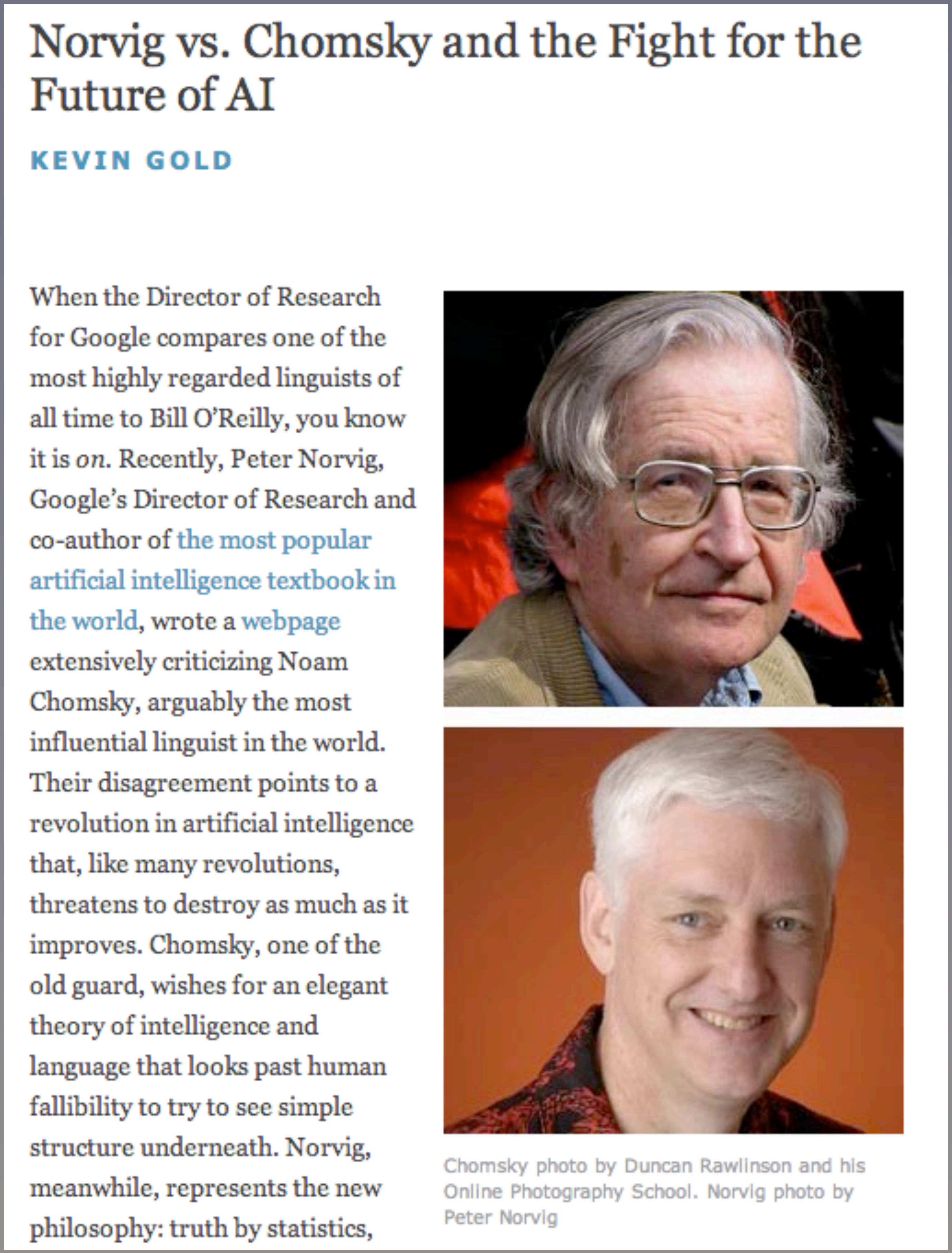
Data-Driven Programs ⬆

Saturday, October 11, 14

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/...



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

Saturday, October 11, 14

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

What Is MapReduce?

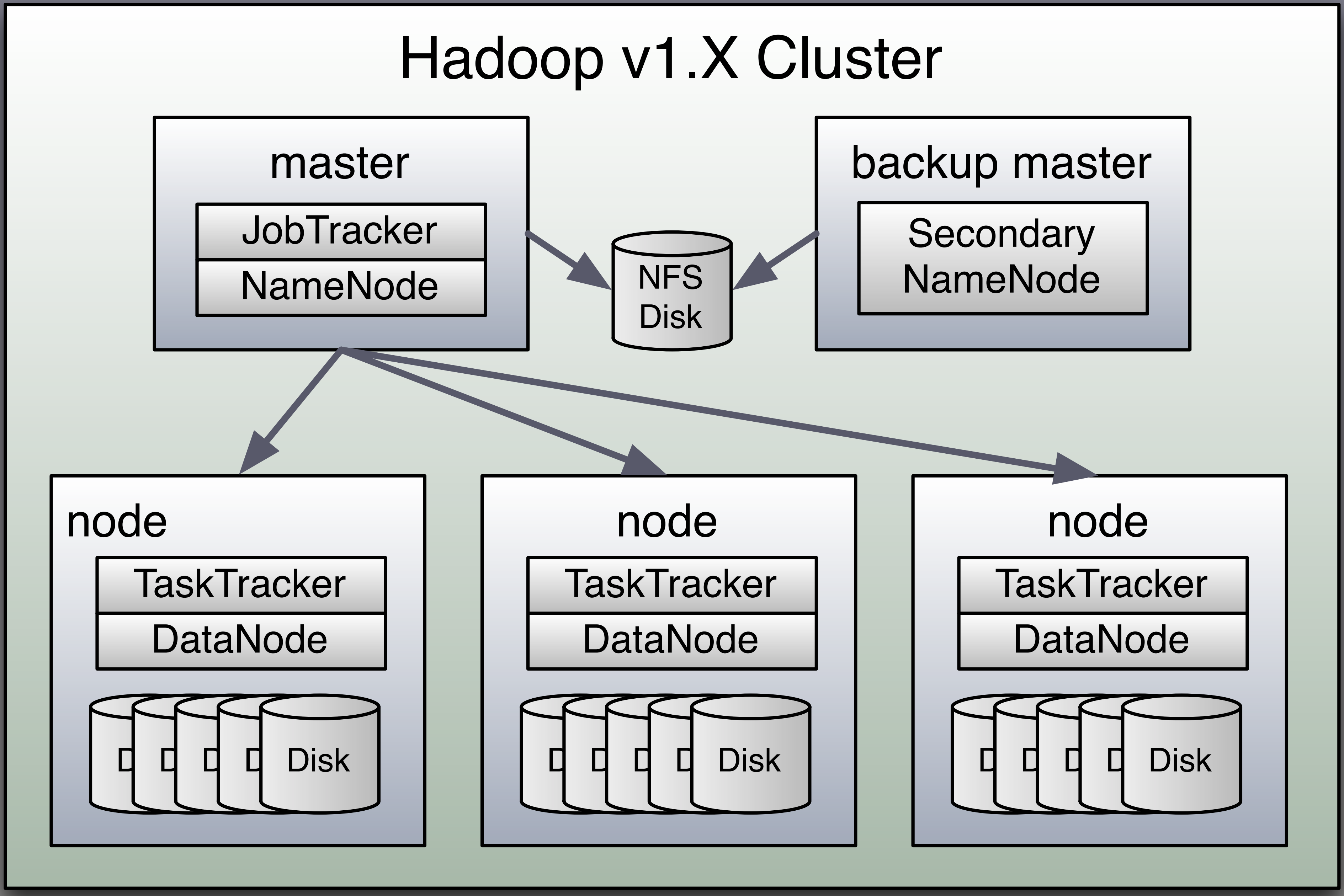
Saturday, October 11, 14

Cloud Gate – “The Bean” – in Millenium Park, Chicago, on a sunny day – with some of my relatives ;)

Hadoop is the dominant copious data platform today.

Saturday, October 11, 14

A Hadoop Cluster



A Hadoop v1.X cluster. (V2.X introduces changes in the master processes, including support for high-availability and federation...). In brief:
JobTracker (JT): Master of submitted MapReduce jobs. Decomposes job into tasks (each a JVM process), often run where the "blocks" of input files are located, to minimize net IO.
NameNode (NN): HDFS (Hadoop Distributed File System) master. Knows all the metadata, like block locations. Writes updates to a shared NFS disk (in V1) for use by the Secondary NameNode.
Secondary NameNode (SNN): periodically merges in-memory HDFS metadata with update log on NFS disk to form new metadata image used when booting the NN and SNN.
TaskTracker: manages each task given to it by the JT.
DataNode: manages the actual blocks it has on the node.
Disks: By default, Hadoop just works with "a bunch of disks" – cheaper and sometimes faster than RAID. Blocks are replicated 3x (default) so most HW failures don't result in data loss.

MapReduce in Hadoop

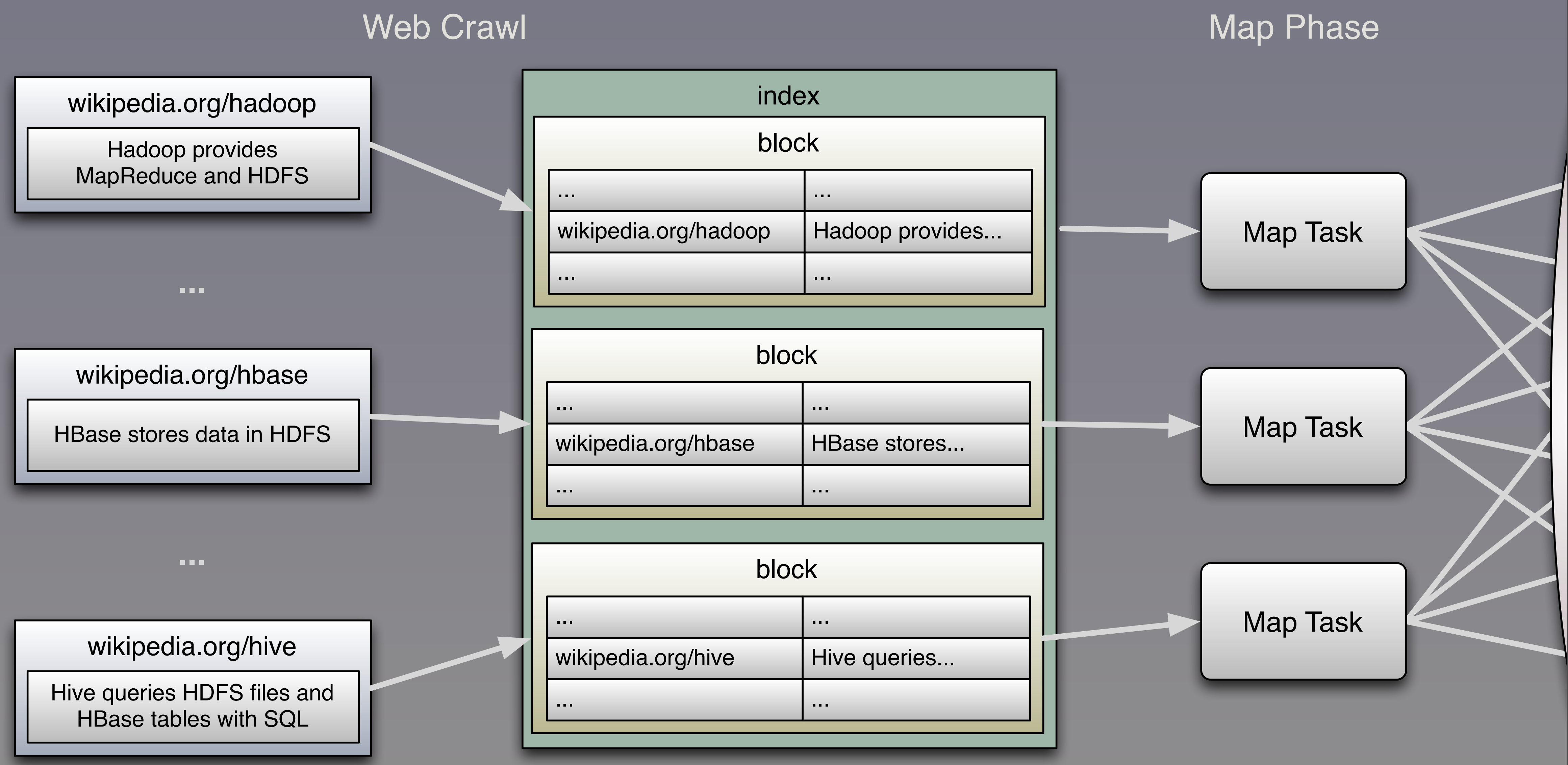
Let’s look at a MapReduce algorithm: Inverted Index.

Used for text/web search.

Saturday, October 11, 14

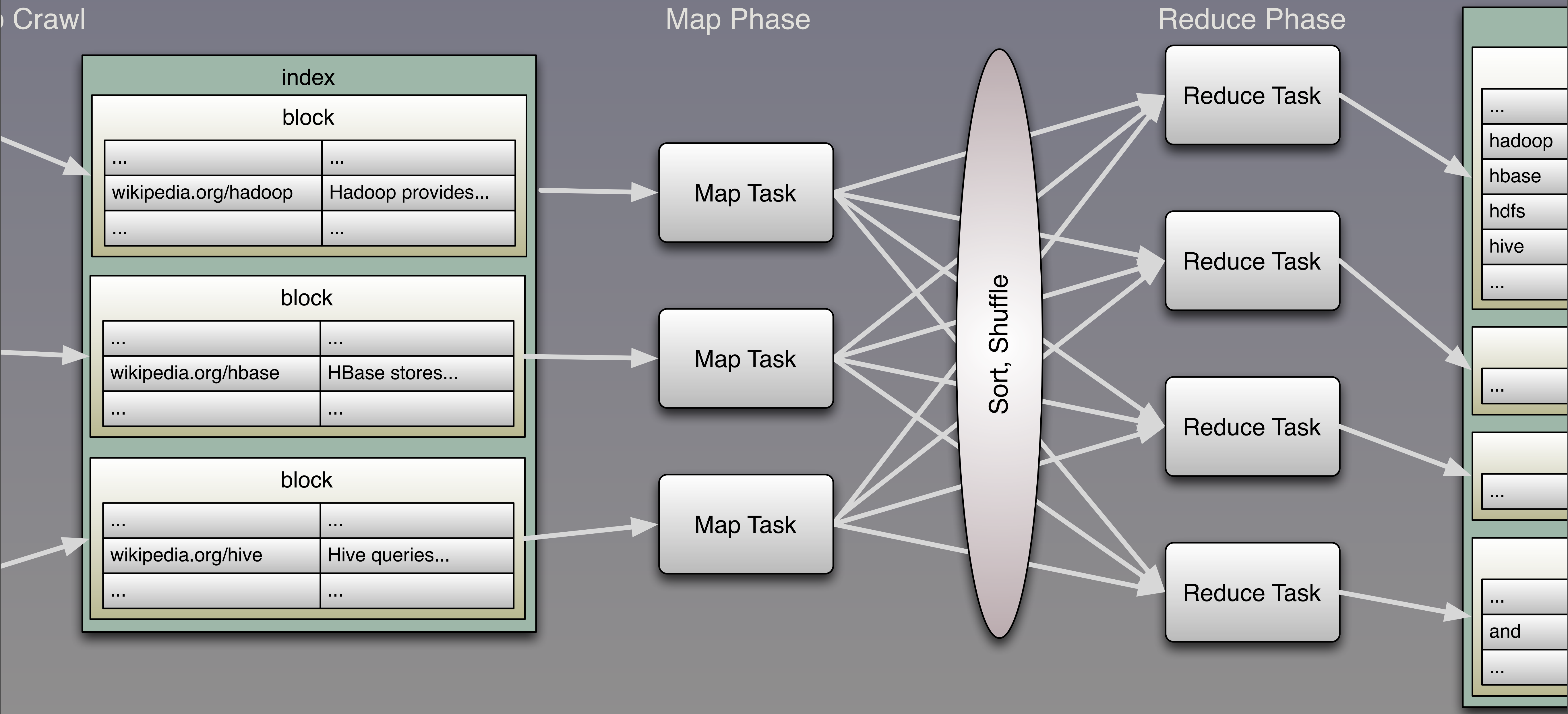
Let’s walk through a simple version of computing an inverted index. Imagine a web crawler has found all docs on the web and stored their URLs and contents in HDFS. Now we’ll index it; build a map from each word to all the docs where it’s found, ordered by term frequency within the docs.

Crawl teh Interwebs



Crawl pages, including Wikipedia. Use the URL as the document id in our first index, and the contents of each document (web page) as the second "column". in our data set.

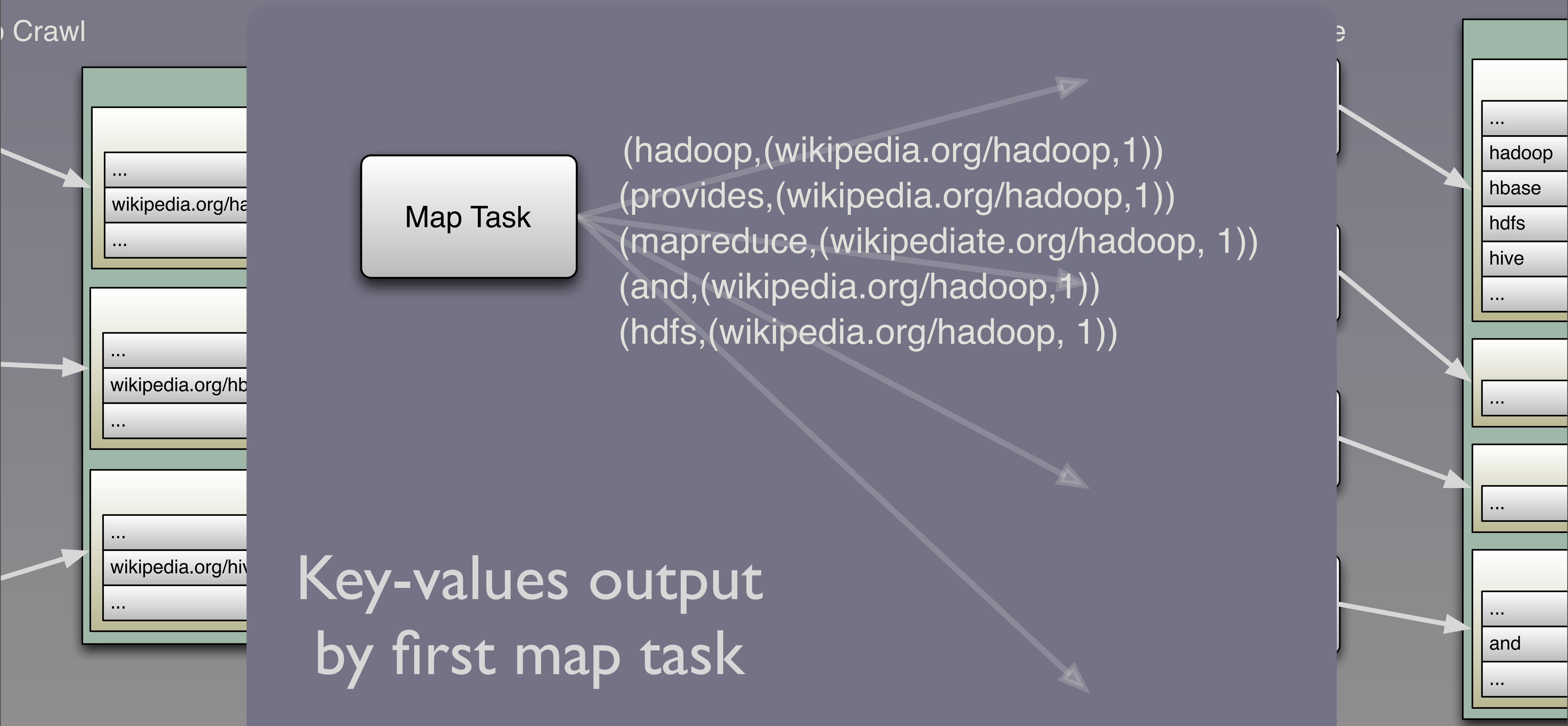
Compute Inverse Index



Saturday, October 11, 14

Now run a MapReduce job, where a separate Map task for each input block will be started. Each map tokenizes the content in to words, counts the words, and outputs key-value pairs...

Compute Inverse Index



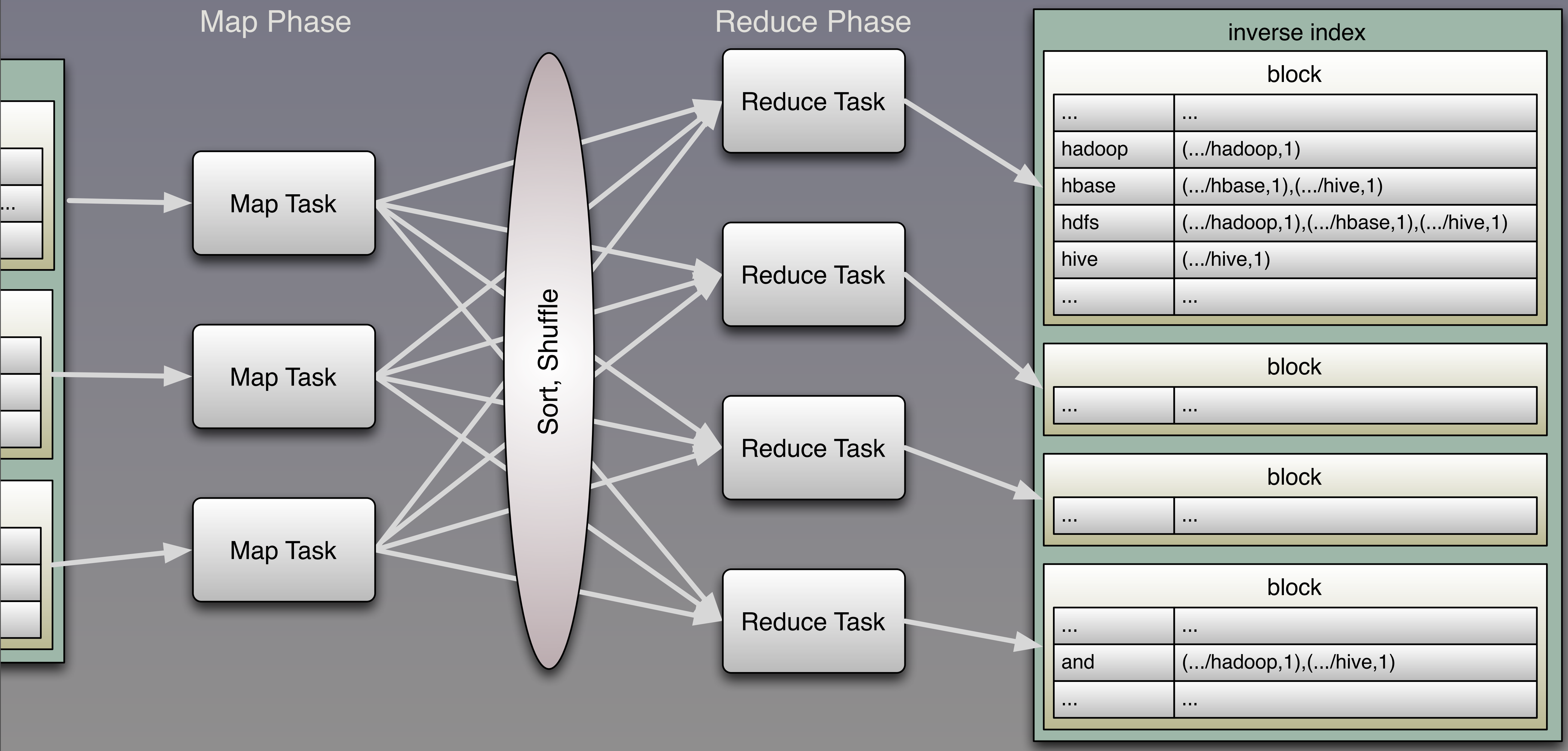
(hadoop,(wikipedia.org/hadoop,1))
(provides,(wikipedia.org/hadoop,1))
(mapreduce,(wikipediate.org/hadoop, 1))
(and,(wikipedia.org/hadoop,1))
(hdfs,(wikipedia.org/hadoop, 1))

Key-values output
by first map task

Saturday, October 11, 14

Now run a MapReduce job, where a separate Map task for each input block will be started. Each map tokenizes the content in to words, counts the words, and outputs key-value pairs...
... Each key is a word that was found and the corresponding value is a tuple of the URL (or other document id) and the count of the words (or alternatively, the frequency within the document). Shown are what the first map task would output (plus other k-v pairs) for the (fake) Wikipedia "Hadoop" page. (Note that we convert to lower case...)

Compute Inverse Index



...	...
hadoop	(.../hadoop,1)
hbase	(.../hbase,1),(.../hive,1)
hdfs	(.../hadoop,1),(.../hbase,1),(.../hive,1)
hive	(.../hive,1)
...	...

...	...

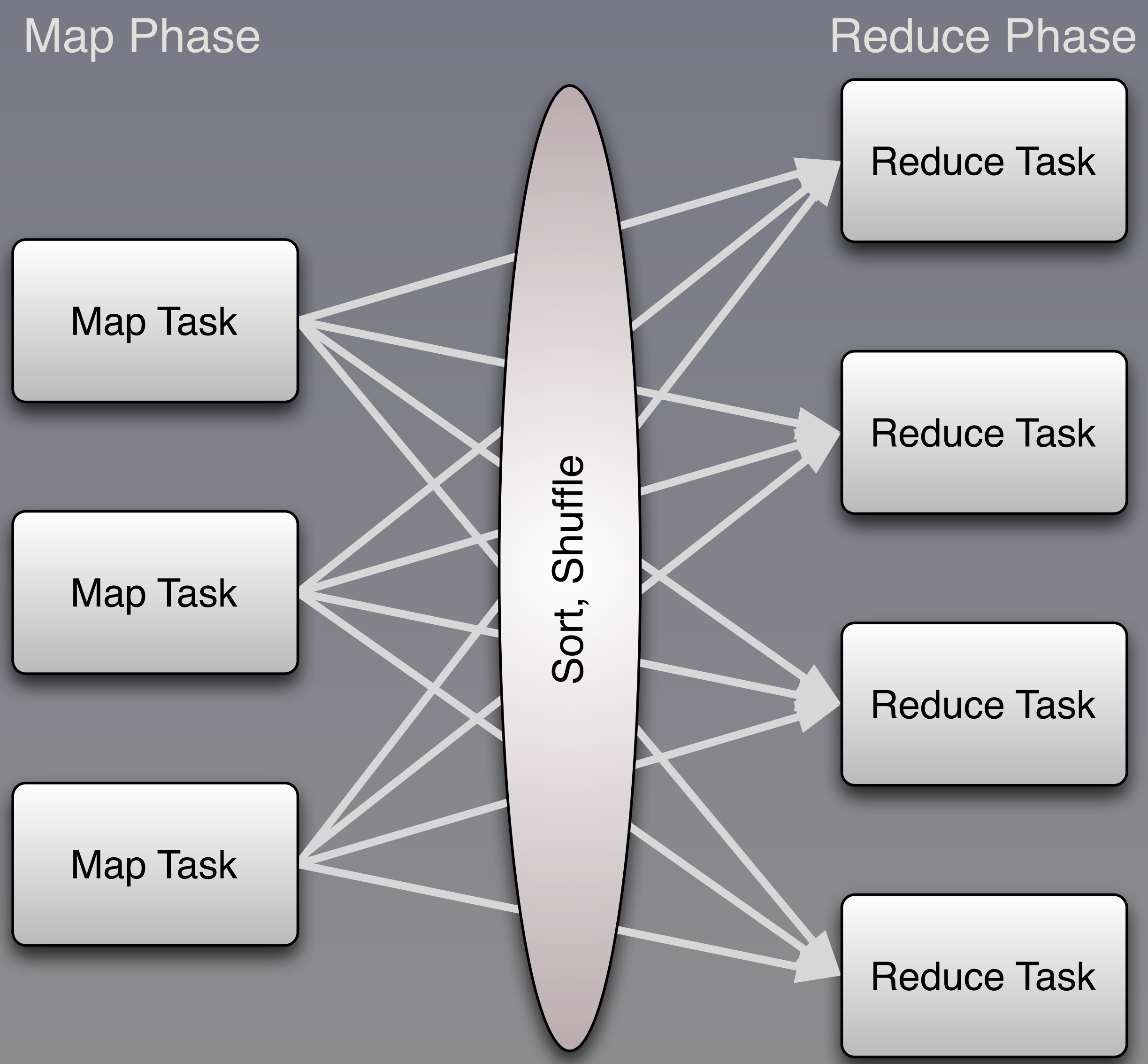
...	...

...	...
and	(.../hadoop,1),(.../hive,1)
...	...

Saturday, October 11, 14

Finally, each reducer will get some range of the keys. There are ways to control this, but we'll just assume that the first reducer got all keys starting with "h" and the last reducer got all the "and" keys. The reducer outputs each word as a key and a list of tuples consisting of the URLs (or doc ids) and the frequency/count of the word in that document, sorted by most frequent first. (All our docs have only one occurrence of any word, so the sort is moot...)

Anatomy: MapReduce Job



Map (or *Flatmap*):

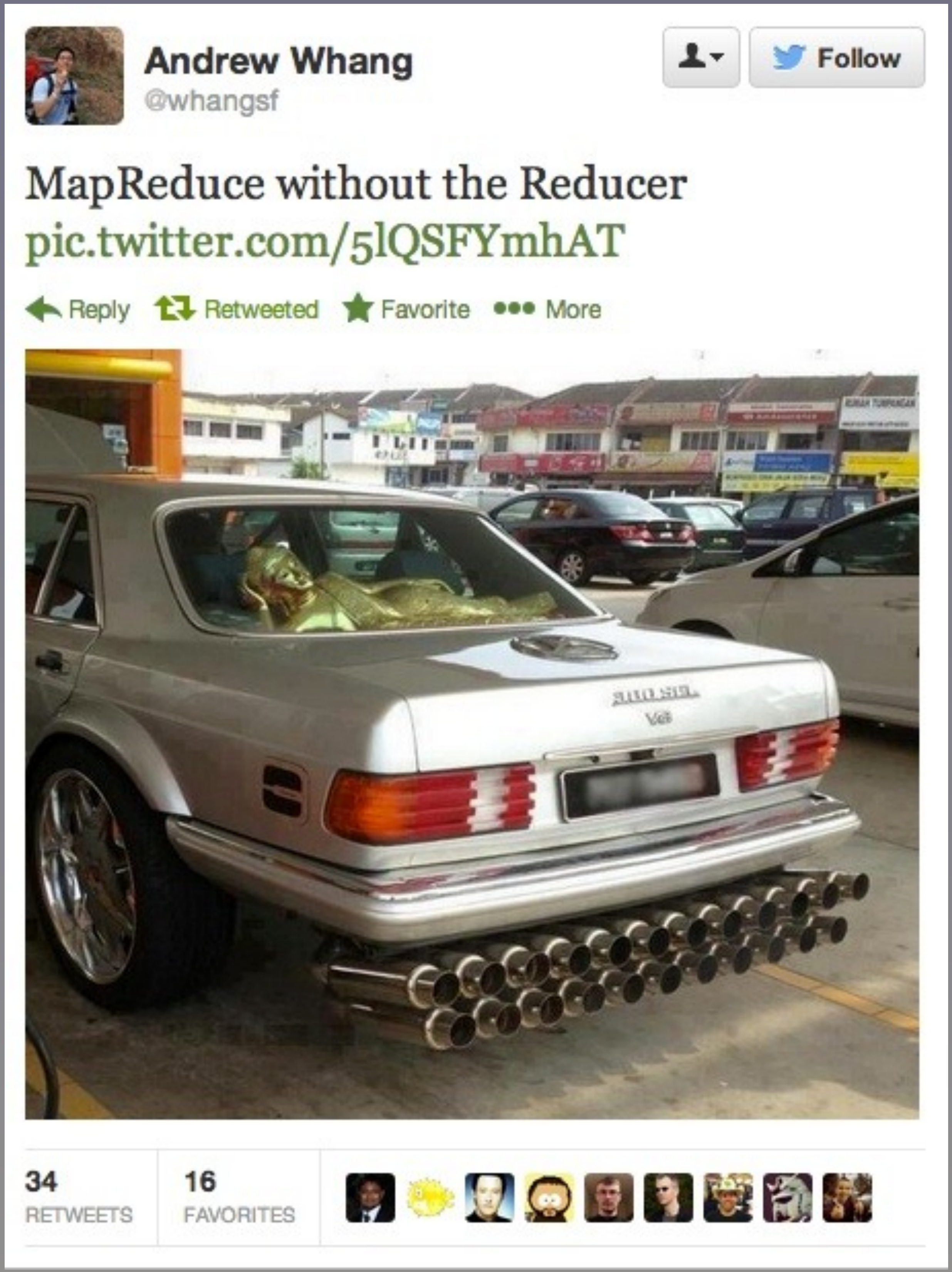
- Transform *one* input to *0-N* outputs.

Reduce:

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

Saturday, October 11, 14

To recap, a true functional/mathematical "map" transforms one input to one output, but this is generalized in MapReduce to be one to 0-N. In other words, it should be "FlatmapReduce"!! The output key-value pairs are distributed to reducers. The "reduce" collects together multiple inputs with the same key into



Andrew Whang
@whangsf

MapReduce without the Reducer
pic.twitter.com/5lQSFYmhAT

Saturday, October 11, 14

Pop Quiz: Do you understand this tweet?

So, MapReduce is a mashup of our friends flatmap and reduce.

Saturday, October 11, 14

Even in this somewhat primitive and coarse-grain framework, our functional data concepts are evident!

Today,
Hadoop is our best, general-purpose tool for horizontal scaling of Copious Data.

MapReduce and Its Discontents

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, Chicago, Feb. 2011.

MapReduce doesn't fit all computation needs. HDFS doesn't fit all storage needs.

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...

Saturday, October 11, 14

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.

Multiple MR jobs required for some algorithms.

Each one flushes its results to disk!

If you have to sequence MR jobs to implement an algorithm, ALL the data is flushed to disk between jobs. There's no in-memory caching of data, leading to huge IO overhead.

MapReduce is designed for offline, batch-mode analytics.

High latency; not suitable for event processing.

Alternatives are emerging to provide event-stream ("real-time") processing.

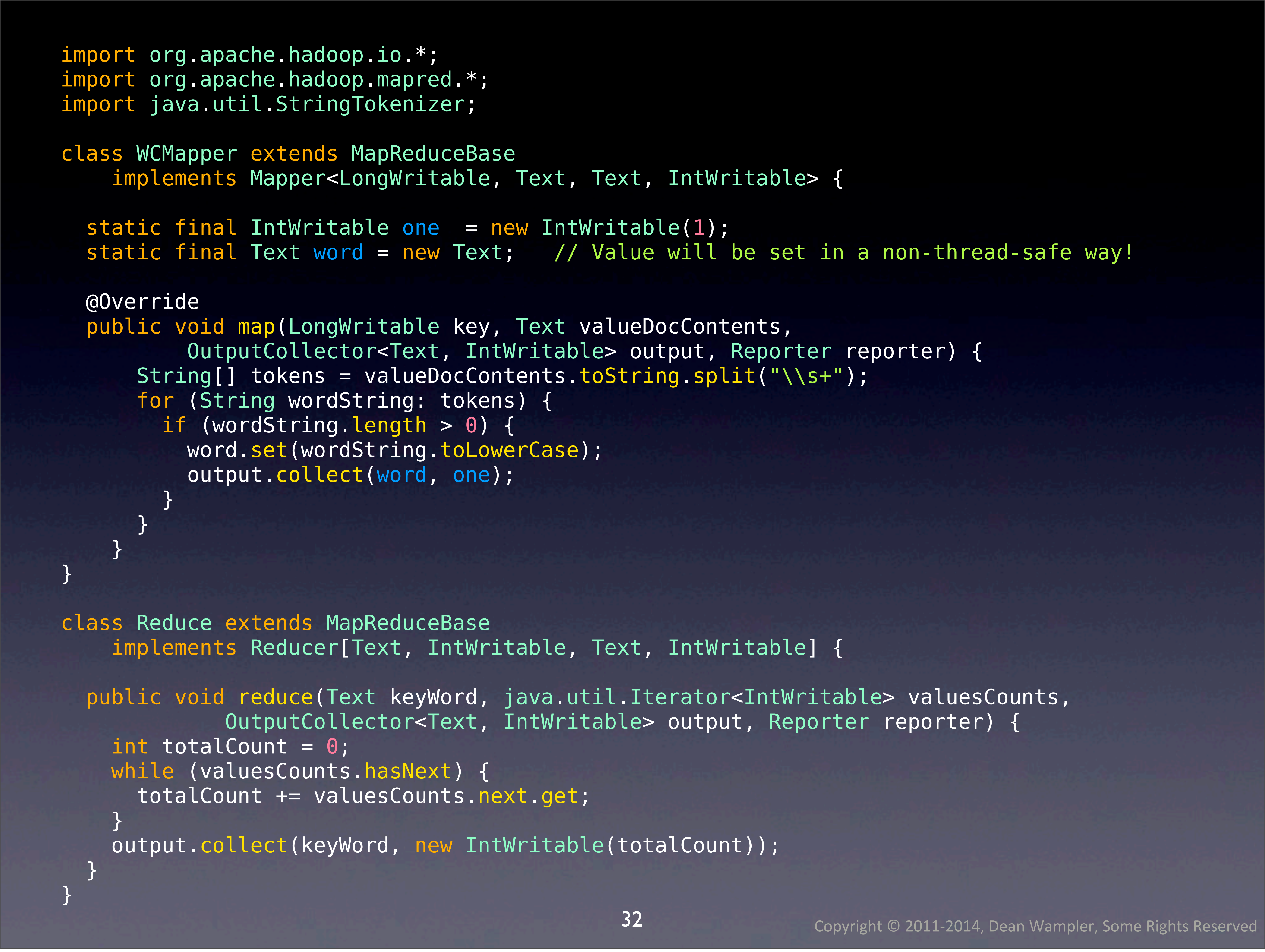
The Hadoop Java API is hard to use.

The Hadoop Java API is even more verbose and tedious to use than it should be.

Let's look at code for a simpler algorithm, Word Count.

(Tokenize as before, but ignore original document locations.)

In Word Count, the mapper just outputs the word-count pairs. We forget about the document URL/id. The reducer gets all word-count pairs for a word from all mappers and outputs each word with its final, global count.



```
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));
  }
}
```

Saturday, October 11, 14

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 dominated by green for types (classes), with relatively few yellow functions that implement actual operations (i.e., do actual work).
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!

```
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));
  }
}
```

The interesting bits

Saturday, October 11, 14

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 dominated by green for types (classes), with relatively few yellow functions that implement actual operations (i.e., do actual work).
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!

```
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));
  }
}
```

The '90s called. They want their EJBs back!

Saturday, October 11, 14

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 dominated by green for types (classes), with relatively few yellow functions that implement actual operations (i.e., do actual work).
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)

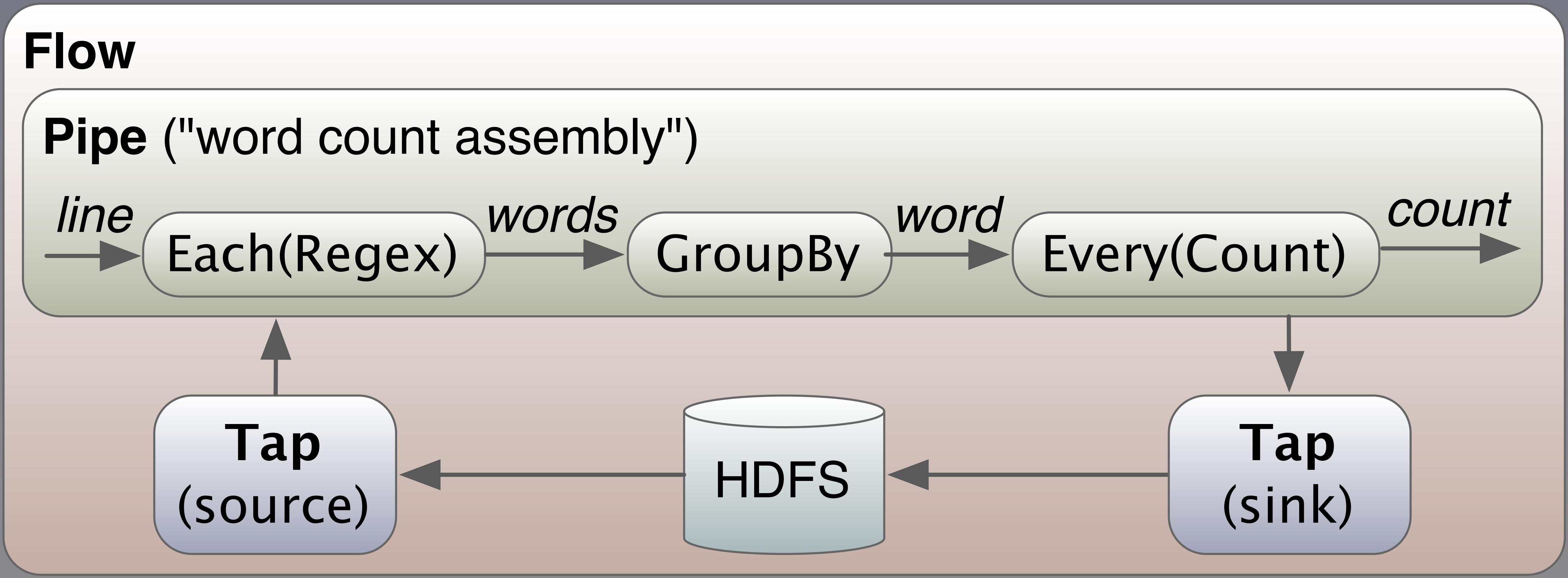
Saturday, October 11, 14

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



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

```java
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, WordCount.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();
  }
}
```

Saturday, October 11, 14

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)

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!!

Saturday, October 11, 14

This Scala code is almost pure domain logic with very little boilerplate. (This is the so-called "Field API" I"m showing. There is a newer Typed API that's slightly different.) 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" model. 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)

Saturday, October 11, 14

http://nathanmarz.com/blog/introducing-cascalog-a-clojure-based-query-language-for-hado.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

Cascalog embeds Datalog-style logic queries. The variables to match are named ?foo.

Use Spark (Not MapReduce)

Saturday, October 11, 14

http://www.spark-project.org/
Spark started as a Berkeley project. recently, the developers launched Databricks to commercialize it, given the growing interest in Spark as a MapReduce replacement. It can run under YARN, the newer Hadoop resource manager (it’s not clear that’s the best strategy, though, vs. using Mesos, another Berkeley project being commercialized by Mesosphere) and Spark can talk to HDFS, the Hadoop Distributed File System.

```
import org.apache.spark.SparkContext

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

Also small and concise!

Saturday, October 11, 14

This spark example is actually closer in a few details, i.e., function names used, to the original Hadoop Java API example, but it cuts down boilerplate to the bare minimum.

Spark is the "Next Generation"

- Recently embraced by Cloudera, MapR, and Hortonworks as a replacement for MapReduce.

Spark

- Distributed computing with in-memory caching.
- ~30x faster than MapReduce (in part due to caching of intermediate data).

Spark also addresses the lack of flexibility for the MapReduce model.

Spark

- Originally designed for machine learning applications.
- Developed by Berkeley AMP.
  - Matei Zaharia

Matei’s graduate work

Use SQL!

Hive, SparkSQL, Impala, Presto, or Lingual

Using SQL when you can! Here are 5 (and growing!) options.

Use SQL when you can!

- Hive: SQL on top of MapReduce.
- SparkSQL: SQL on Spark.
- Impala & Presto: HiveQL with new, faster back ends.
- Lingual: ANSI 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. http://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920 for Presto. Impala & Presto are relatively new.

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;
```

... and similarly for the other SQL tools.

Saturday, October 11, 14

This is how you could implement word count in Hive. We're using some Hive built-in functions for tokenizing words in each "line", the one "column" in the docs table, etc., etc.
Lingual is similarly, but because it's more ANSI-compliant, the example would be much different.

We're in the era where *The SQL Strikes Back!*

(with apologies to George Lucas...)

IT shops realize that NoSQL is useful and all, but people really, Really, REALLY love SQL. So, it's making a big comeback. You can see it in Hadoop, in SQL-like APIs for some "NoSQL" DBs, e.g., Cassandra and MongoDB's Javascript-based query language, as well as "NewSQL" DBs.

Combinators

Photo: The defunct Esquire movie theater on Oak St., off the “Magnificent Mile”, in Chicago. Now completely gone!

Why were the Scala, Clojure, and SQL solutions so concise and appealing??

Data problems are fundamentally Mathematics!

evanmiller.org/mathematical-hacker.html

A blog post about how developers ignore mathematics at their peril!

Category Theory

- Monads - Structure.
  - Abstracting over collections.
  - Control flow and mutability containment.

Monads generalize the properties of containers, like lists and maps, such as applying a function to each element and returning a new instance of the same container type. This also applies to encapsulations of state transformations and “principled mutability”, as used in Haskell.

Category Theory

- Monoids, Groups, Rings, etc.
  - Abstracting over addition, subtraction, multiplication, and division.

Monoid: Addition

- (a + b) + (c + d) for some a, b, c, d.
- "Add All the Things", Avi Bryant, StrangeLoop 2013.

infoq.com/presentations/abstract-algebra-analytics

For an explanation of this slide, see this great presentation by Avi Bryant at StrangeLoop 2013 on generalizing addition (monoids).

Linear Algebra

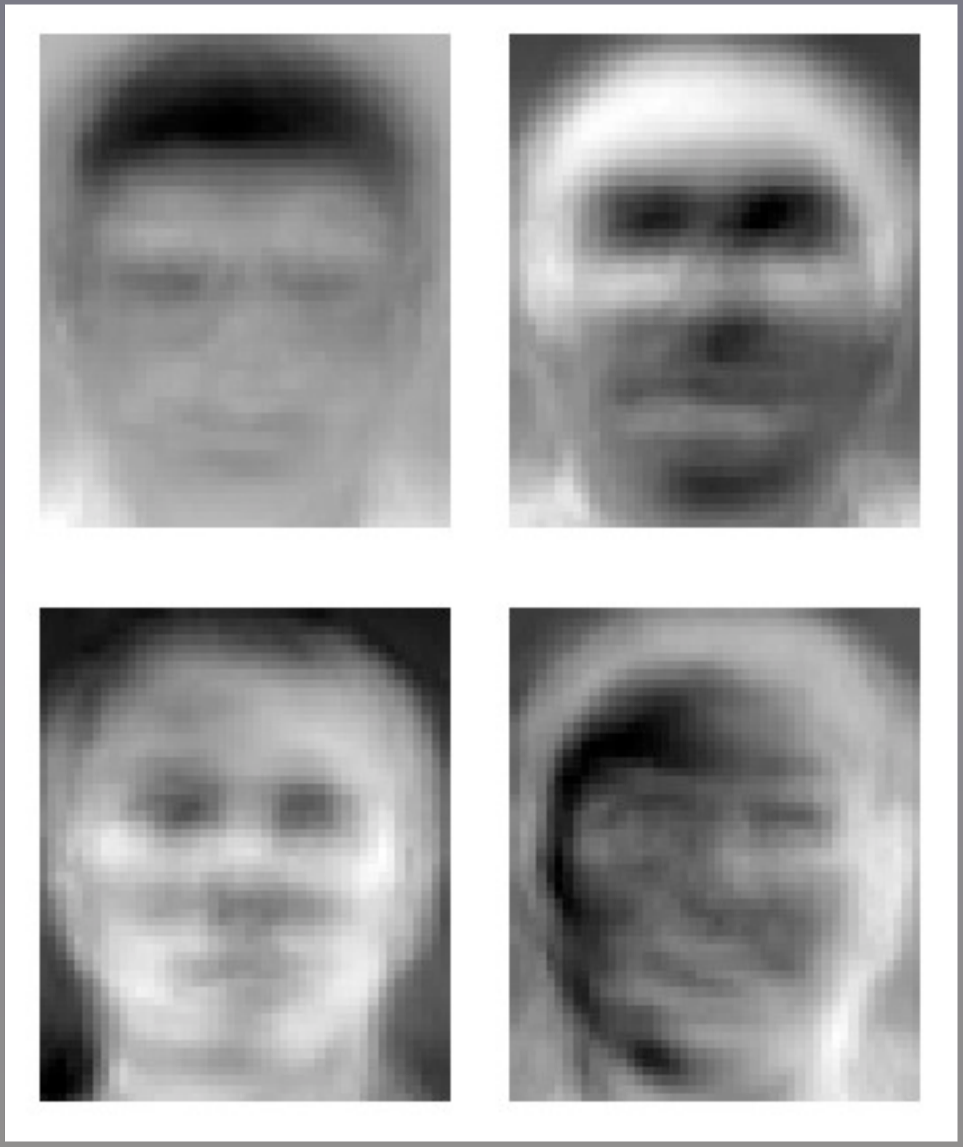
- Singular Value Decomposition and Principal Component Analysis
  - Essential tools in machine learning.

$$Av = \lambda v$$

The equation is for a related concept, eigenvector decomposition.

Example: Eigenfaces

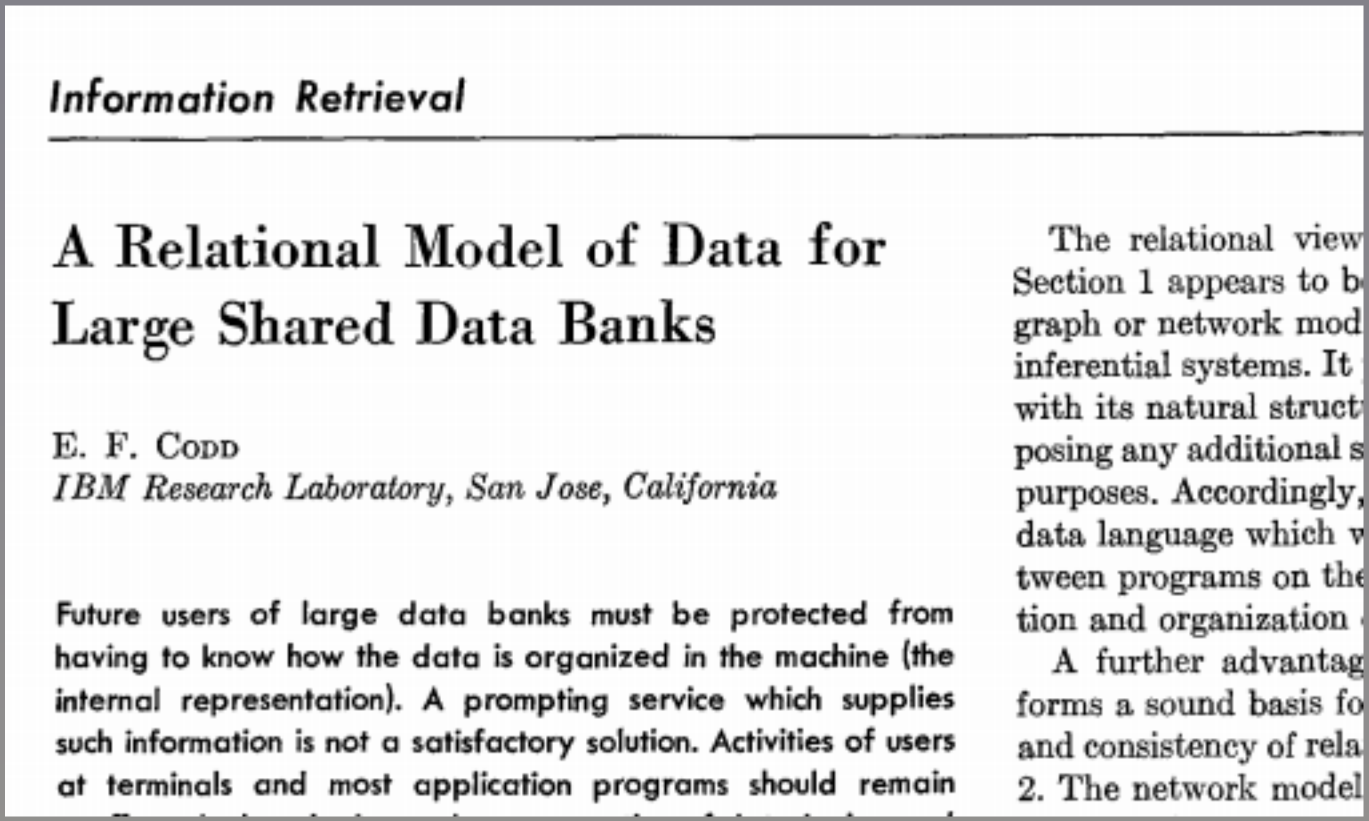
- Represent images as vectors.
- Solve for "modes".
- Top N modes approx. faces!



http://en.wikipedia.org/wiki/File:Eigenfaces.png

Set Theory and First-Order Logic

- Relational Model.
- Data organized into tuples, grouped by relations.



Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain

The relational view
Section 1 appears to b
graph or network mod
inferential systems. It
with its natural struct
posing any additional s
purposes. Accordingly,
data language which w
tween programs on the
tion and organization
A further advantag
forms a sound basis fo
and consistency of rela
2. The network model

http://dl.acm.org/citation.cfm?doid=362384.362685

Formulated by Codd in ’69. Most systems don’t follow it exactly, like allowing identical records, where set elements are unique. Codd’s original model didn’t support NULLs either (“unknown”), but he later proposed a revision to allow them.

Set Theory and First-Order Logic

- Relational Model.
  - Most RDBMSs deviate from RM.

Formulated by Codd in ’69. Most systems don’t follow it exactly, like allowing identical records, where set elements are unique. Codd’s original model didn’t support NULLs either (“unknown”), but he later proposed a revision to allow them.

What are Combinators?

- Functions that are side-effect free.
  - They get all their information from their inputs and write all their work to their outputs.

Invented by Mathematicians...

Let's look at 4 relational operators and the corresponding functional combinators.

See, for example, the discussions in "Database in Depth" and "SQL and Relational Theory, Second Edition", both by C.J. Date (O'Reilly)

Recall our Word Counts:

```
CREATE TABLE word_counts (
  word   CHARACTER(64),
  count  INTEGER);
```

(ANSI SQL syntax)

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

(Scala)

Saturday, October 11, 14

Our word_counts table from before, using ANSI SQL syntax this time.
The corresponding Scala might be any kind of collection, e.g., a List. Here, I'll use a Stream, which is a lazy collection useful for large data structures like I/O...
Note that it's a stream of a two-element tuple, a String (for the word) and an Int (for the count).

Restrict

```
SELECT * FROM word_counts
WHERE word = 'Chicago';
```

vs.

```
word_counts.filter {
  case (word, count) =>
    word == "Chicago"
}
```

Saturday, October 11, 14

For the Scala example, assume word_counts is a collection (List, Vector, etc.) of 2-element tuples. The case match in the anonymous function passed to filter is a way of conveniently assigning variables to each element of the tuple, here “word” and “count”. Then I filter on only certain word values.

Project

```
SELECT word FROM word_counts;
```

vs.

```
word_counts.map {
  case (word, count) =>
    word
}
```

Here, I just return the words in each record or Scala tuple.

Join

```
CREATE TABLE dictionary (
  word       CHARACTER(64),
  definition CHARACTER(256));
```

Table for join examples.

First, we need something to join with; let's use a dictionary of word definitions.

Join - SQL

```
SELECT w.word, d.definition
FROM   word_counts AS w
       dictionary AS d
WHERE  w.word = d.word;
```

Saturday, October 11, 14

Here is the SQL join that gives us the words and their definitions. (side note: Hive doesn’t support this “inferred” join syntax; you have to use a more explicit JOIN … ON … syntax.)

Join - Scalding

```
val word_counts =
  Csv("/path/…", ('wword, 'count)).read
val definitions =
  Csv("/path/…", ('dword, 'definition)).read

word_counts
  .joinWithLarger('wword -> 'dword,
    dictionary)
  .project('wword, 'definition)
```

The Scala collections library doesn't have a join combinator. We would have to build up something that understands the data, such as exploiting sort order. This is a case where a large-scale data system will implement expensive operations, where a general-purpose programming library might not. So, I'm using a Scalding example.

Join

```
SELECT w.word, d.definition
FROM   word_counts AS w
       dictionary AS d
WHERE  w.word = d.word;
```

vs.

```
...
word_counts
  .joinWithLarger('wword -> 'dword,
    dictionary)
  .project('wword, 'definition)
```

Now shown together, with some of the Scalding setup code removed.

Joins are expensive. Your data system needs to exploit optimizations.

Group By

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

vs.

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

How many words appeared once, twice, 3 times, ..., N-times? Order this list descending.
I'm back to the Scala library (as opposed to Scalding). The code inputs a collections of tuples, (word,count) and groups by count. This creates a map with the count as the key and a list of the words as the value.
Next we convert this to a list of tuples (count,List(words)) and map it to a list of tuples with the (count, size of List(words)), then finally sort descending by the list sizes.

Example

```
scala> val word_counts = List(
("a", 1), ("b", 2), ("c", 3),
("d", 2), ("e", 2), ("f", 3))

scala> val out = word_counts.groupBy {
  case (word, count) => count
}.toList.map {
  case (count, words) => (count, words.size)
}.sortBy {
 case (count, size) => -size
}

out: List[(Int,Int)] =
  List((2,3), (3,2), (1,1))
```

Saturday, October 11, 14

Here's a simple example you can run in the Scala REPL (prompts are "scala>").

We could go on, but you get the point. Declarative, functional combinators are a natural tool for data.

SQL vs. FP

- SQL
  - Optimized for data operations.
- FP
  - Turing complete.
  - More combinators.
  - First class functions!

A drawback of SQL is that it doesn't provide first class functions, so (depending on the system) you're limited to those that are built-in or UDFs (user-defined funcs) that you can write and add. FP languages make this easy!!

FP to the Rescue!

Outside my condo window one Sunday morning...

Popular Claim:

Multicore concurrency is driving FP adoption.

We've all heard this. In fact, this is how I got interested in FP.

My Claim:

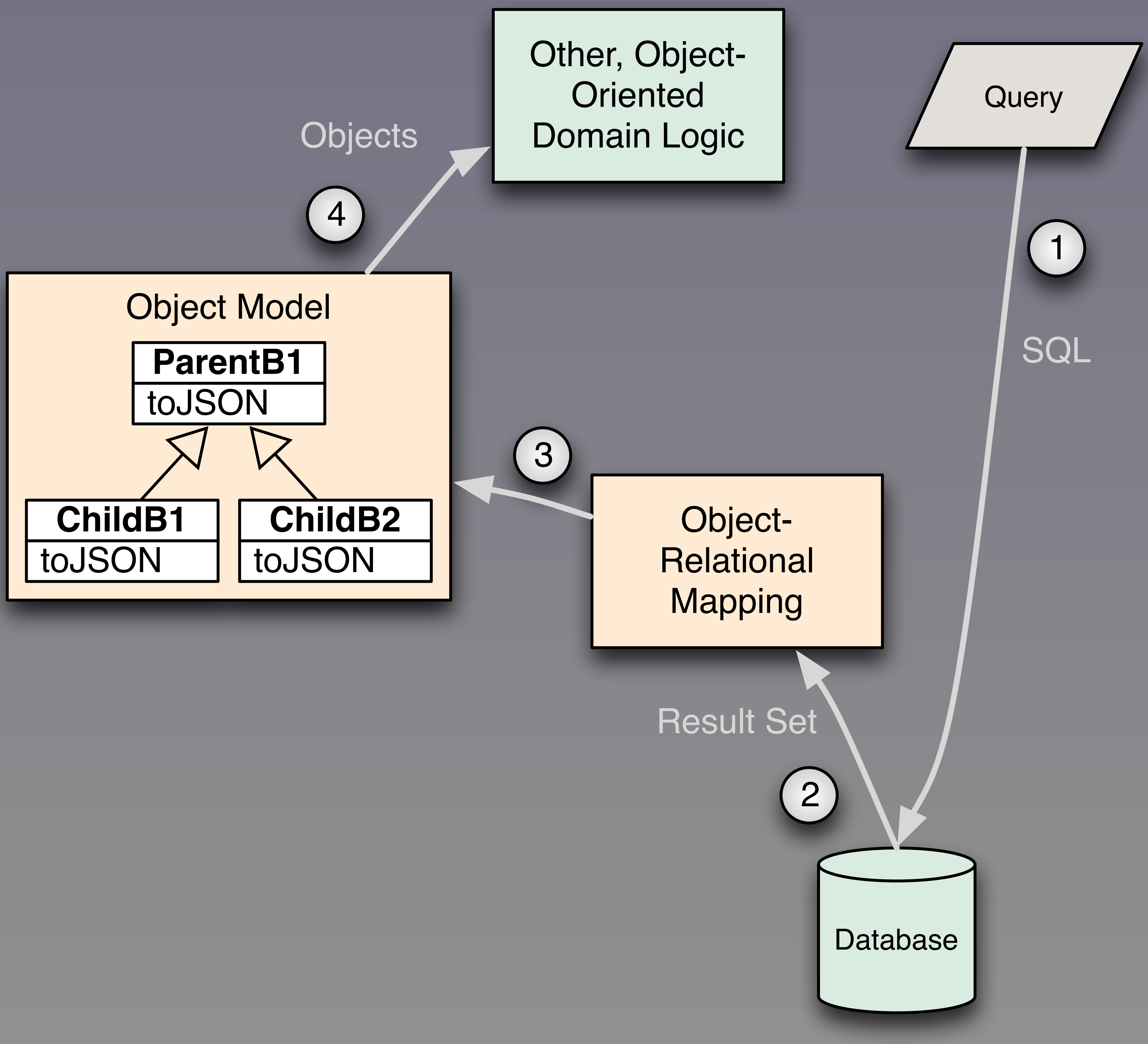
Data will drive the next wave of widespread FP adoption.

Even today, most developers get by without understanding concurrency. Many will just use an Actor or Reactive model to “solve” their problems. I think more devs will have to learn how to work with data at scale and that fact will drive them to FP. This will be the next wave.

Data Architectures

Saturday, October 11, 14

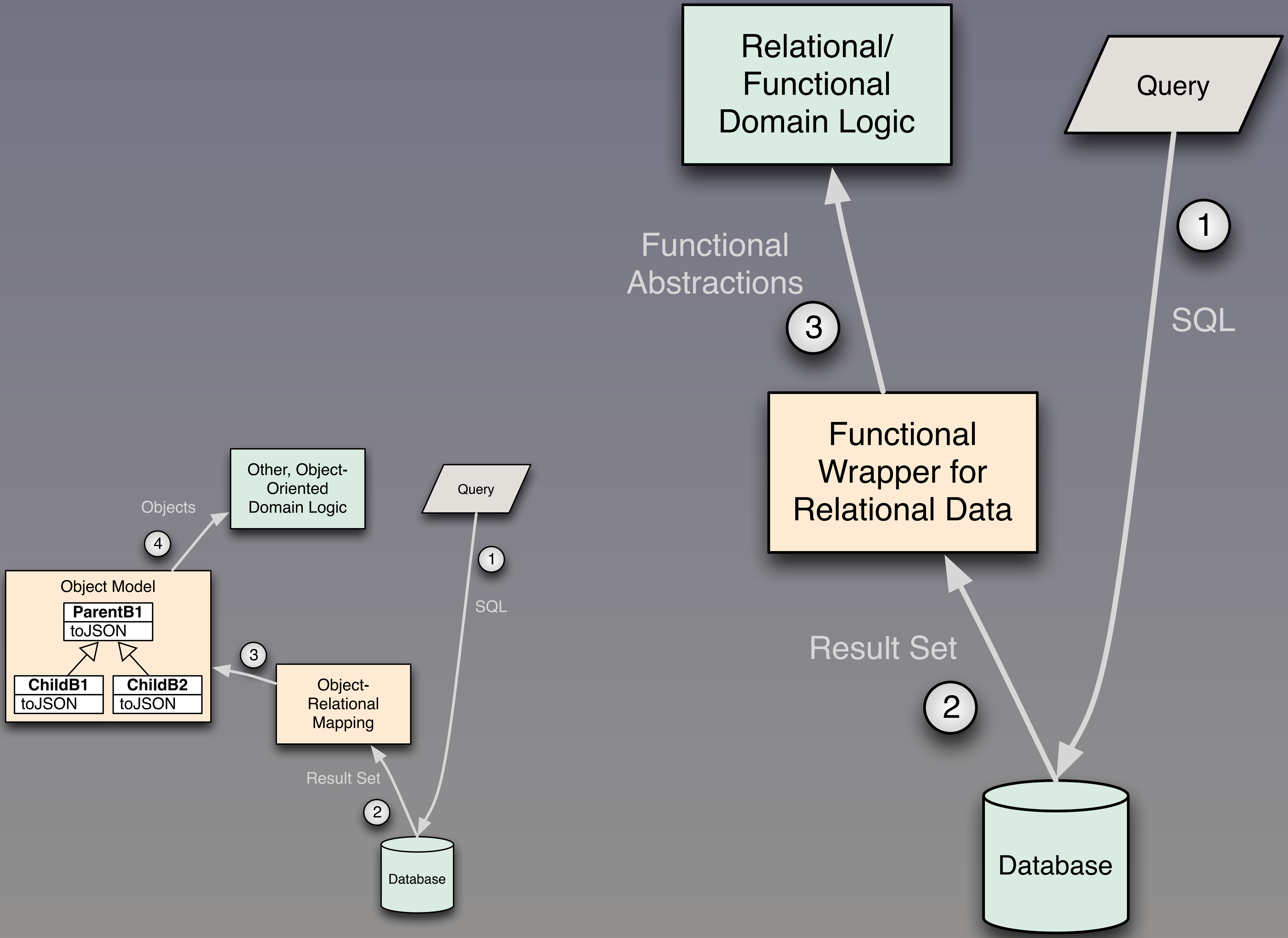
What should software architectures look like for these kinds of systems?
Photo: Two famous 19th Century Buildings in Chicago.



Saturday, October 11, 14

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…)

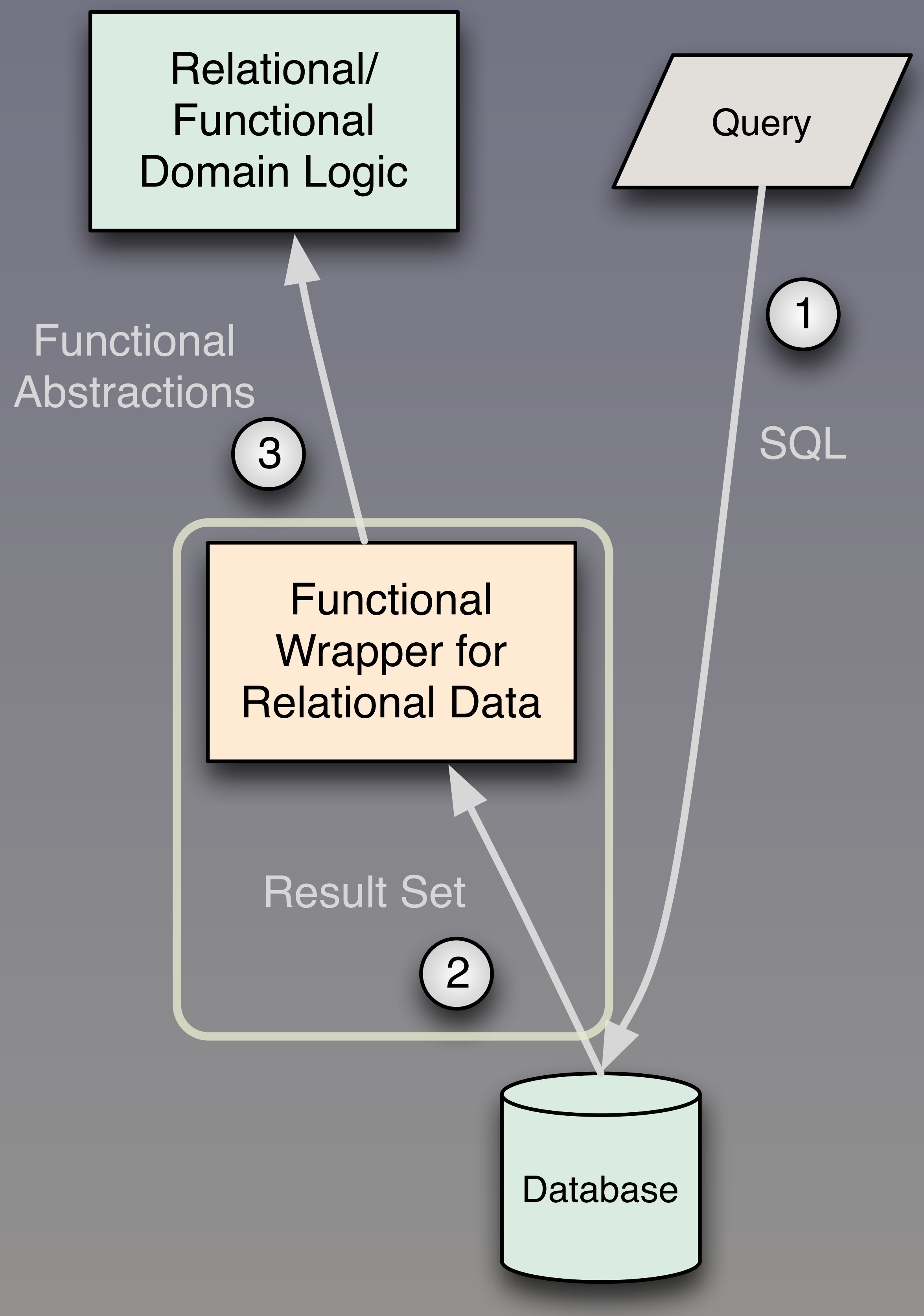
The toJSON methods are there because we often convert these object graphs back into fundamental structures, such as the maps and arrays of JSON so we can send them to the browser!



Saturday, October 11, 14

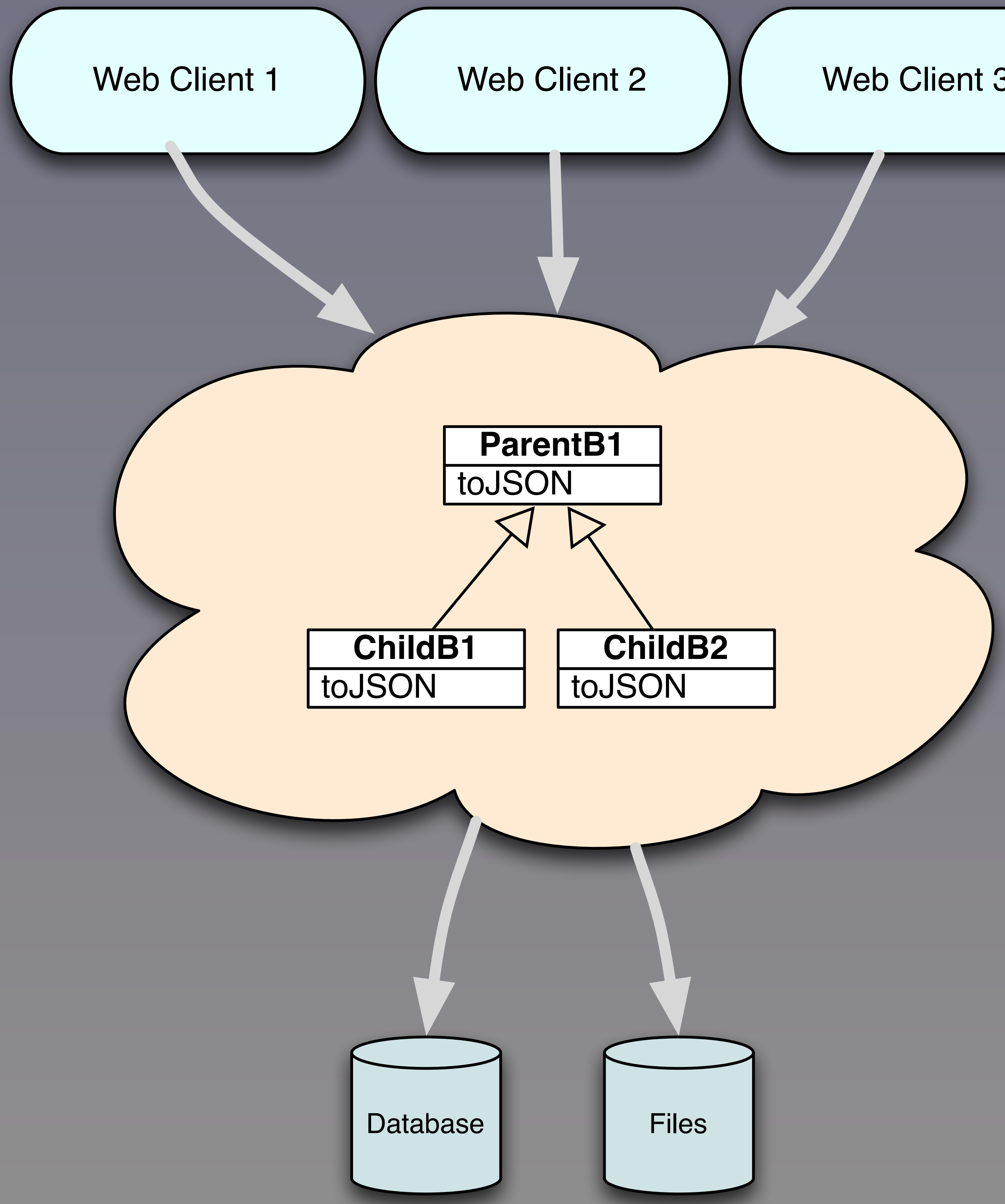
But the traditional systems are a poor fit for this new world: 1) they add too much overhead in computation (the ORM layer, etc.) and memory (to store the objects). Most of what we do with data is mathematical transformation, so we're far more productive (and runtime efficient) if we embrace fundamental 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
  - ...


Relational/ Functional Domain Logic
Query
1
SQL
Functional Abstractions
3
Functional Wrapper for Relational Data
Result Set
2
Database

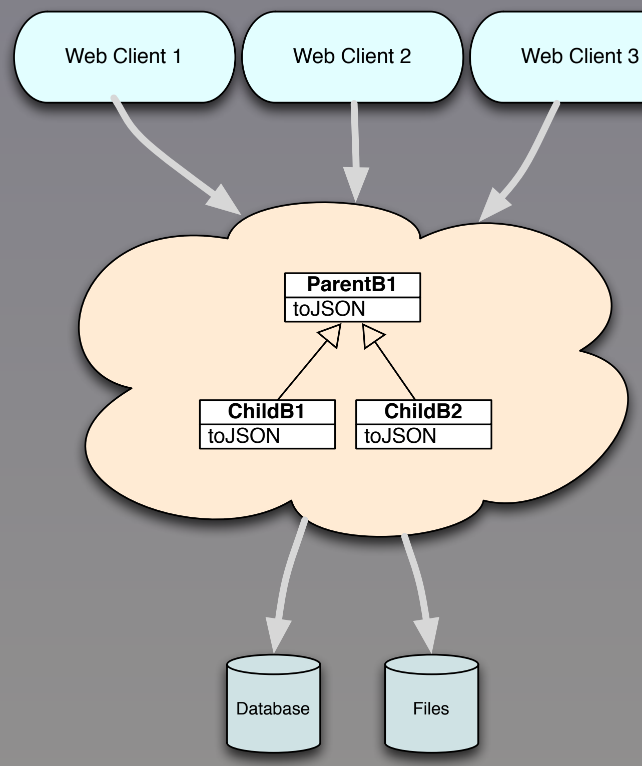
Saturday, October 11, 14

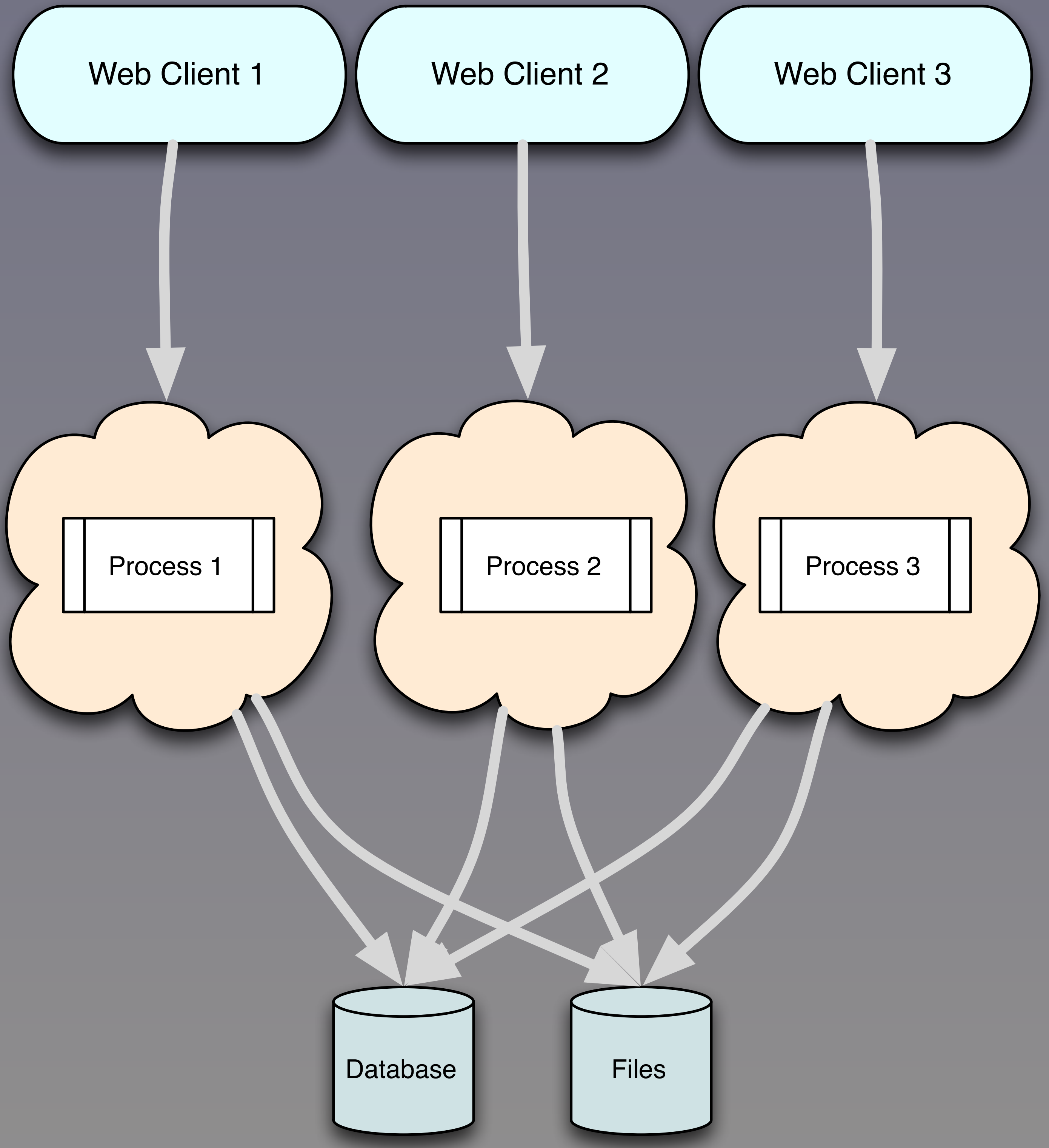
But the traditional systems are a poor fit for this new world: 1) they add too much overhead in computation (the ORM layer, etc.) and memory (to store the objects). Most of what we do with data is mathematical transformation, so we're far more productive (and runtime efficient) if we embrace fundamental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



Saturday, October 11, 14

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.

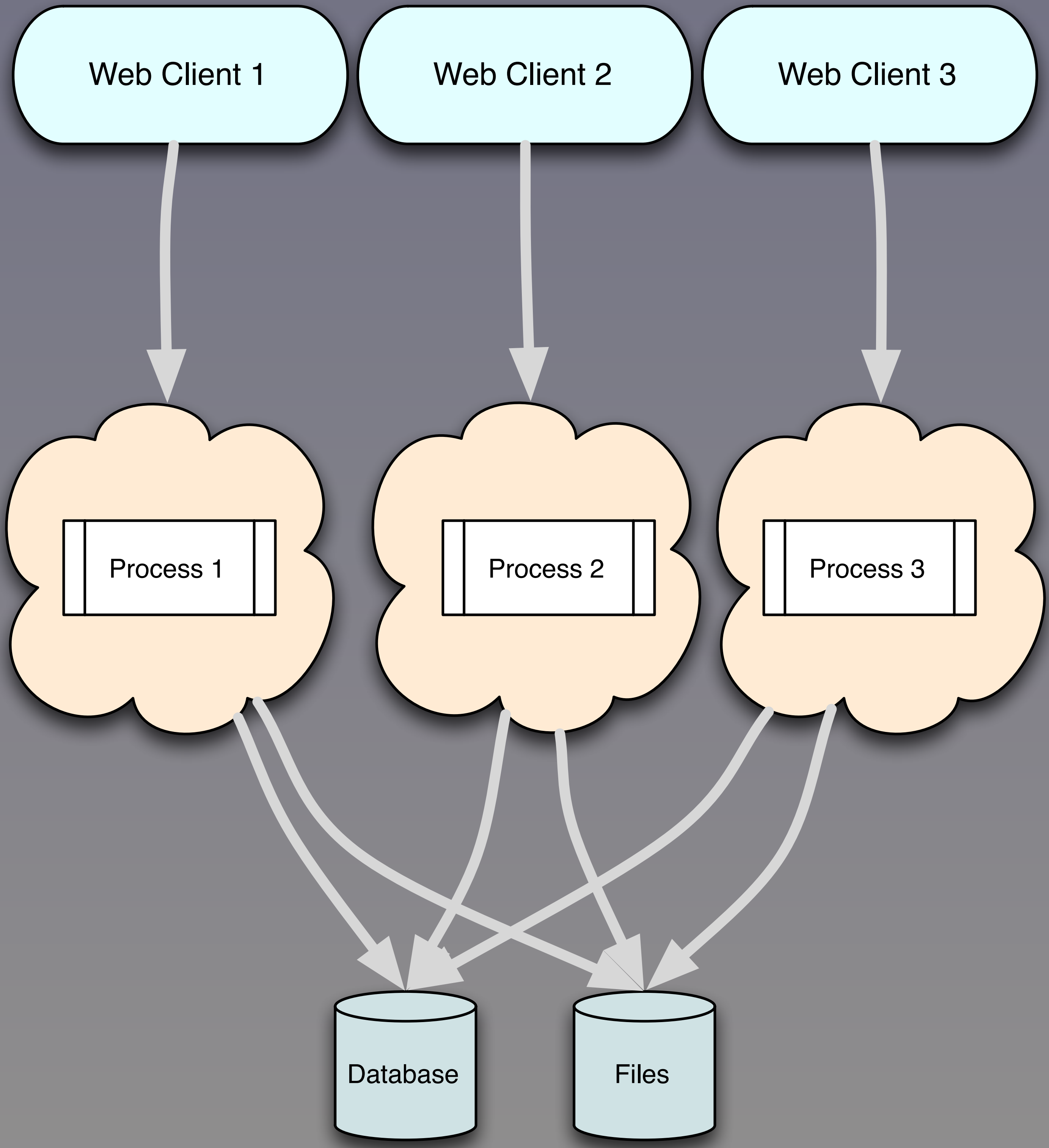




Saturday, October 11, 14

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 ⬆



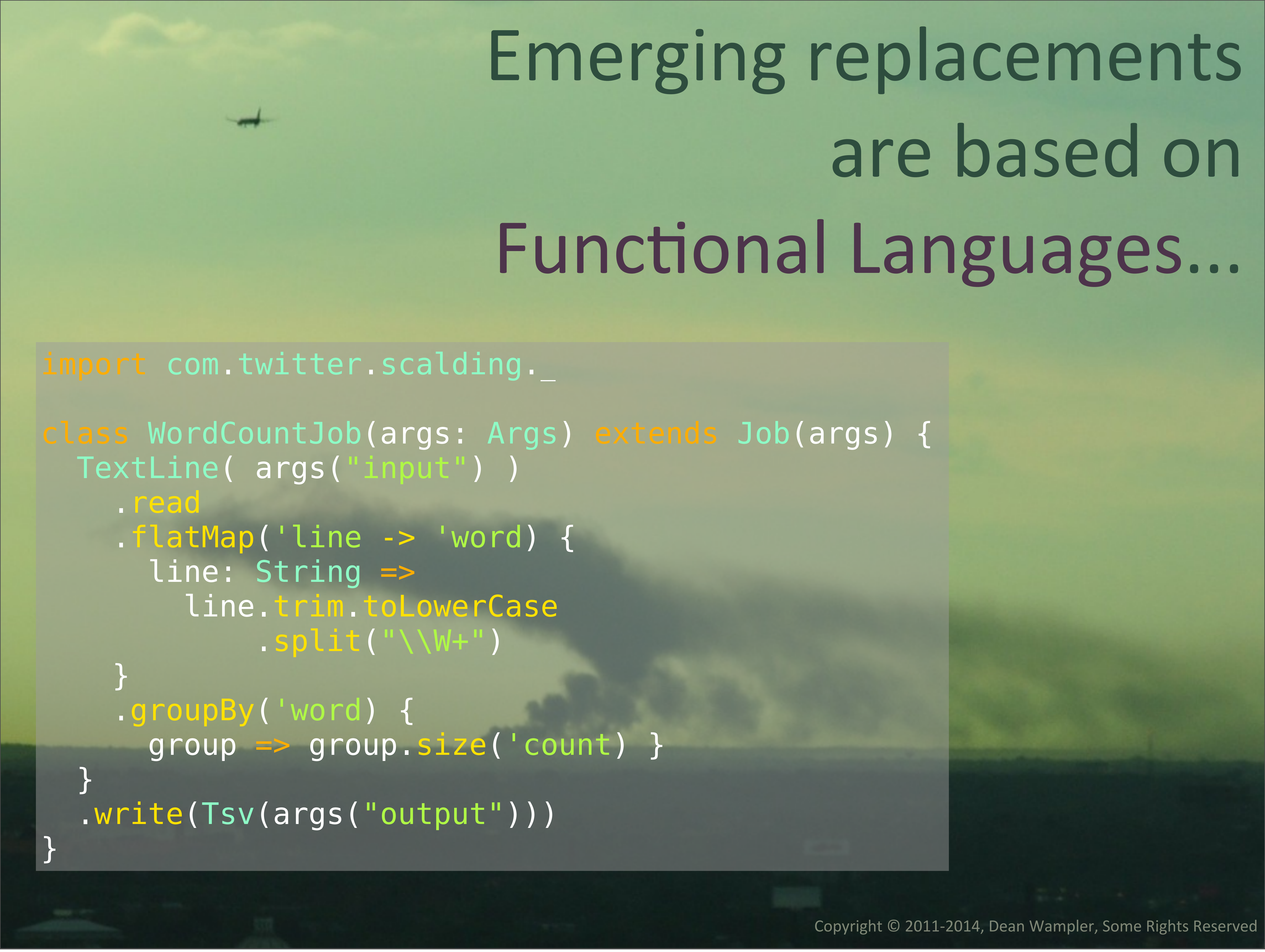
Saturday, October 11, 14

And this structure better fits the trends I outlined at the beginning of the talk.

Hadoop MapReduce is the Enterprise Java Beans of our time.

Saturday, October 11, 14

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...



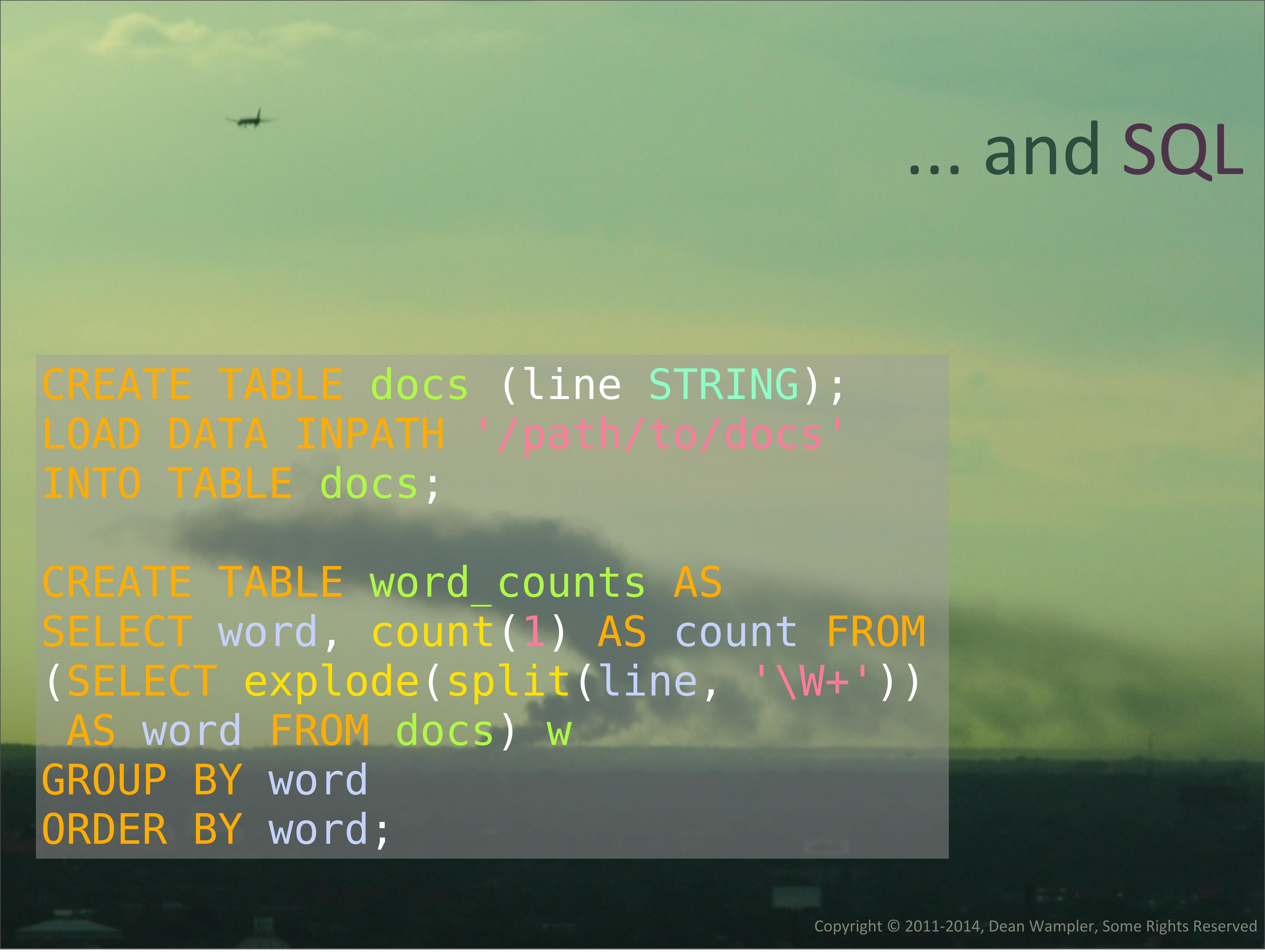
Emerging replacements are based on Functional Languages...

```
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")))
}
```

Saturday, October 11, 14

We've seen a lot of issues with MapReduce. Already, alternatives like Spark for general use and Storm for event stream processing, are gaining traction. Also, special, purpose-built replacements, like Impala, are popular.
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.



… and 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;
```

Saturday, October 11, 14

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...

Questions?

October 29, 2014
BigData Techcon San Francisco
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

All pictures Copyright © Dean Wampler, 2011-2014, Some Rights Reserved. All other content is free to use, but attribution is requested.
Photo: Building in fog on Michigan Avenue

Bonus Slides

Hive

- SQL dialect.
- Used MapReduce back end (so annoying latency). Now running on Tez.
- First SQL on Hadoop.
- Developed by Facebook.

http://hive.apache.org

SparkSQL

- SQL integrated with Spark's API.
- Also work with Hive tables.
- Excellent performance.

Impala

- HiveQL front end (subset).
- 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.

Presto

- HiveQL front end.
- Java back end.
- Provides up to 10-100x performance improvement!
- Developed by Facebook.

See http://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920. However, this was just announced a few weeks ago (at the time of this writing), so it’s new, but already in wide use at Facebook.

Lingual

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

http://www.cascading.org/lingual/ A relatively new "API" for Cascading, still in Beta. Because Cascading runs on MapReduce (there's also a standalone "local mode" for small jobs and development), it will have the same perf. characteristics as Hive and other MR-based tools.