Data-Intensive Information Processing Applications — Session #2

# Hadoop: Nuts and Bolts

**Jordan Boyd-Graber**
University of Maryland

Tuesday, February 10, 2011

# Last Class

- Registration
- Sign up for mailing list
- Complete usage agreement (so you get on the cluster)
- Notecards
  - Difficult class
  - Real-world examples
- How to sort a list of numbers

# Naive Way to Sort Numbers

- Mapper: Identity Mapper (just emit everything)
- Reducer: Output everything
- Postprocess: Merge results (why?)

| | | | |
|---|---|---|---|
| 1 | 4 | 15 | 9 |
| 2 | 65 | 35 | 89 |
| 97 | 79 | 323 | 8462 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 9 | 15 | 35 | 65 | 79 | ... |

# Better Way to Sort Numbers

- Assume *K* reducers
- Sample small fraction of data to guess at *K* evenly spaced numbers ($p_1$, $p_2$, $p_3$, $p_4$, ... $p_{K-1}$)
- Create new partitioner(x)
  - $x < p_1$: reducer 1
  - $p_i <= x < p_{i+1}$: reducer i
  - $p_K <= x$ : reducer K
- Concatenate output
- Sorted 1TB of data in 209 seconds (first OSS / Java win)

# This class: Hadoop Programs

- Configuring / Setting up Jobs
- Representing Data
- What happens underneath
- How to write / test / debug Hadoop programs

# Hadoop Programming

- Remember “strong Java programming” as pre-requisite?
- But this course is *not* about programming!
  - Focus on “thinking at scale” and algorithm design
  - We’ll expect you to pick up Hadoop (quickly) along the way
- How do I learn Hadoop?
  - This session: brief overview
  - White’s book
  - RTFM, RTFC(!)

Source: Wikipedia (Mahout)

# Basic Hadoop API

- Mapper
  - void map(K1 key, V1 value, Context context)
  - context.write(k, v) – Used to emit intermediate results
- Reducer/Combiner
  - void reduce(K2 key, Iterable<V2> values, Context context)
  - context.write(k, v) – Used to emit results
- Partitioner
  - int getPartition(K2 key, V2 value, int numPartitions)
  - Returns the partition assignment
- Job / Configuration
  - Specifies the mappers / reducers / combiners / partitioners
  - Sets options (command line or from XML)

# Data Types in Hadoop

| | |
|---|---|
| ***Writable*** | Defines a de/serialization protocol. Every data type in Hadoop is a Writable. |
| ↑ | |
| ***WritableComprable*** | Defines a sort order.  All keys must be of this type (but not values). |
| ↑ | |
| **IntWritable**<br>**LongWritable**<br>**Text**<br>**…** | Concrete classes for different data types. |
| **SequenceFiles** | Binary encoding of a sequence of key/ value pairs |

# Where Can I Find Writables?

- Hadoop
- Cloud9: edu.umd.cloud9.io
  - Arrays
  - HashMap
  - Pairs
  - Tuples

*Table 4-6. Writable wrapper classes for Java primitives*

| Java primitive | Writable implementation | Serialized size (bytes) |
|---|---|---|
| boolean | BooleanWritable | 1 |
| byte | ByteWritable | 1 |
| int | IntWritable | 4 |
| | VIntWritable | 1–5 |
| float | FloatWritable | 4 |
| long | LongWritable | 8 |
| | VLongWritable | 1–9 |

# "Hello World": Word Count

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
        Emit(term, value);
```

# Three Gotchas

- Avoid object creation, at all costs
- Execution framework reuses value in reducer (Clone)
- Passing parameters into mappers and reducers
  - DistributedCache for larger (static) data
  - Configuration object for smaller parameters (unit tests?)

# Complex Data Types in Hadoop

- How do you implement complex data types?
- The easiest way:
  - Encoded it as Text, e.g., (a, b) = "a:b"
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish
- The hard way:
  - Define a custom implementation of WritableComprable
  - Must implement: readFields, write, compareTo, hashCode
  - Computationally efficient, but slow for rapid prototyping
- Alternatives:
  - Cloud$^9$ offers two other choices: Tuple and JSON
  - (Actually, not that useful in practice)
  - Google: Protocol Buffers

# Protocol Buffers

- Developed by Google
- Now open source
- Arbitrary data types
- Compiled into language of your choice
  - Python
  - C++
  - Java
  - (Other languages by folks outside of Google)
- Data are represented by compact byte streams

# Why use Protocol Buffers

- Ad hoc data types are under-specified
  - 10.2010
    - Is it a date?
    - A number?
    - A string?
- Reading in data is often CPU-bound
  - Parsing CSV / XML is faster with two CPUs than one
  - Note: goes against CS accepted wisdom
- Cross-platform
  - OS
  - Programming language
- Extensible
- Scales well (Google has multi-gigabyte protocol buffers)

# Why not use Protocol Buffers

- Needs libraries to be installed for every language
- One additional thing to compile
- Not human readable
- Needs up front investment to design data structures (somtimes a good thing)

# Protocol Buffers: Source

```
package tutorial;
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

**Metadata to generate Source code**

**Name of protocol buffer**

**Typed data**

**Discrete data**

**Sub-type definition**

**Sub-type use**

# Protobuffs in your favorite language

- Compile the source into code:

```
package com.example.tutorial;

public final class AddressBookProtos {
  pr
  pu

  }
  pu
...
```

```
package com.example.tutorial;
 public static com.example.tutorial.AddressBookProtos.Person.PhoneNumber
pa
rseF
```

```
public void writeTo(com.google.protobuf.CodedOutputStream output)
                    throws java.io.IOException {
    getSerializedSize();
    if (((bitField0_ & 0x00000001) == 0x00000001)) {
      output.writeBytes(1, getNameBytes());
    }
    if (((bitField0_ & 0x00000002) == 0x00000002)) {
      output.writeInt32(2, id_);
    }
    if (((bitField0_ & 0x00000004) == 0x00000004)) {
      output.writeBytes(3, getEmailBytes());
...
```

- Get IO, serialization, type checking, and access for free

# Steps for writing protcol buffer

- Design data structure
- Compile protocol buffer:

**protoc addressbook.proto --java_out=. --cpp_out=. --python_out=.**

- Create source code using protocol buffers
- Compile your code, include PB library
- Deploy

```
for (Person.PhoneNumber phoneNumber :
person.getPhoneList()) {
        switch (phoneNumber.getType()) {
          case MOBILE:
            System.out.print("  Mobile phone #: ");
            break;
          case HOME:
            System.out.print("  Home phone #: ");
            break;
          case WORK:
            System.out.print("  Work phone #: ");
            break;
        }
```

# Protocol Buffers – Moral

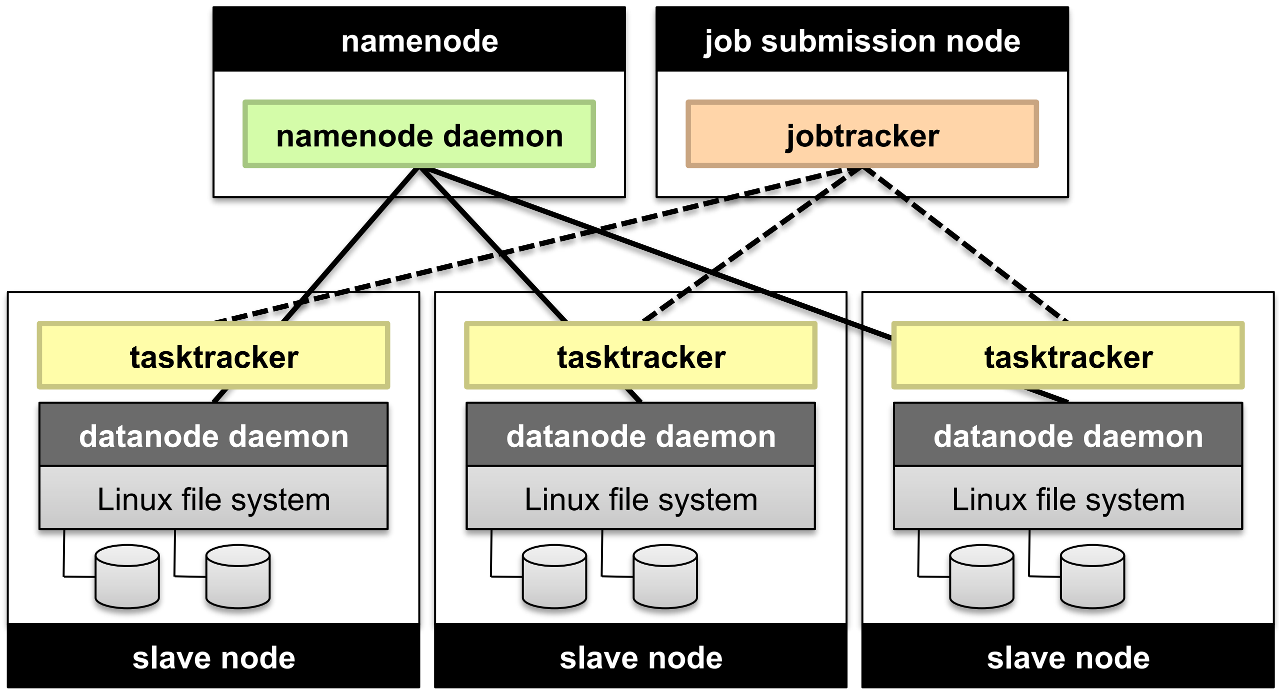
- Crossplatform method to store data
- Good support in MapReduce
  - Google: All messages assumed to be protocol buffers
  - Hadoop: Package called Elephant-Bird (Twitter)
- Use when
  - Not in control of the data you get
  - Writing in many different programming langauges
  - Raw data need not be human readable
  - Complex projects
- Welcome and encouraged to use them for class (but not required)


Source: Wikipedia

# Basic Cluster Components

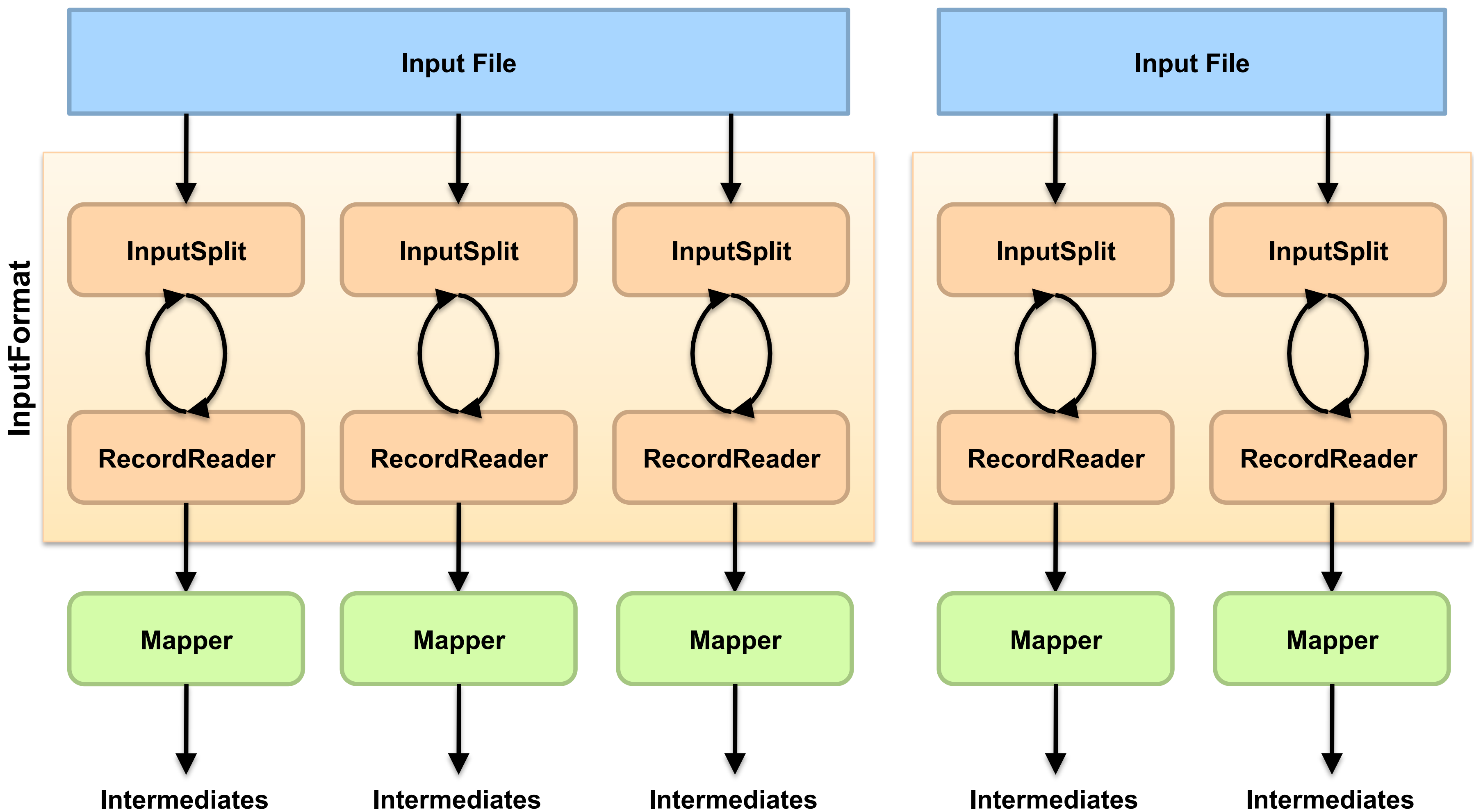
- One of each:
  - Namenode (NN)
  - Jobtracker (JT)
- Set of each per slave machine:
  - Tasktracker (TT)
  - Datanode (DN)

# Putting everything together…

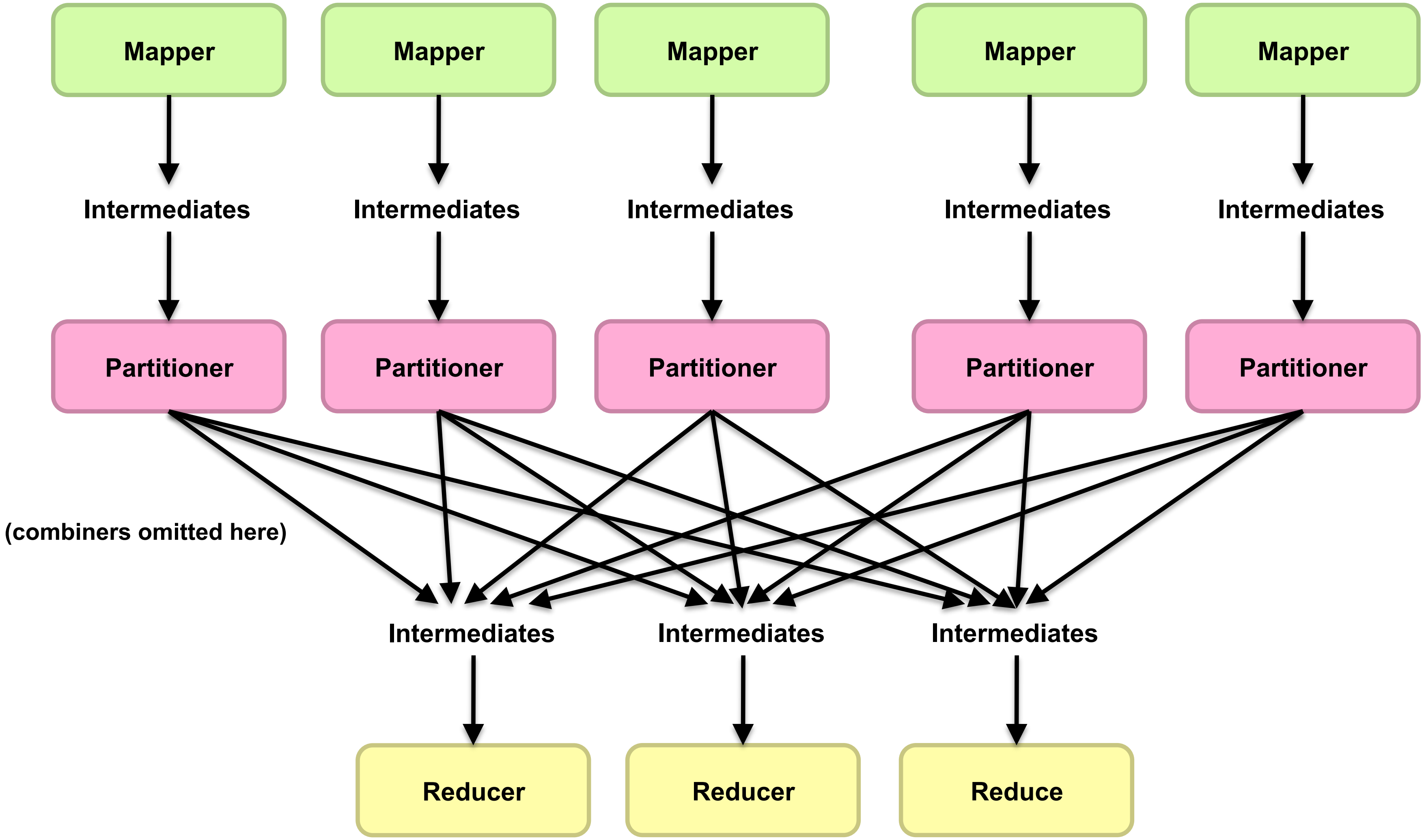


# Anatomy of a Job

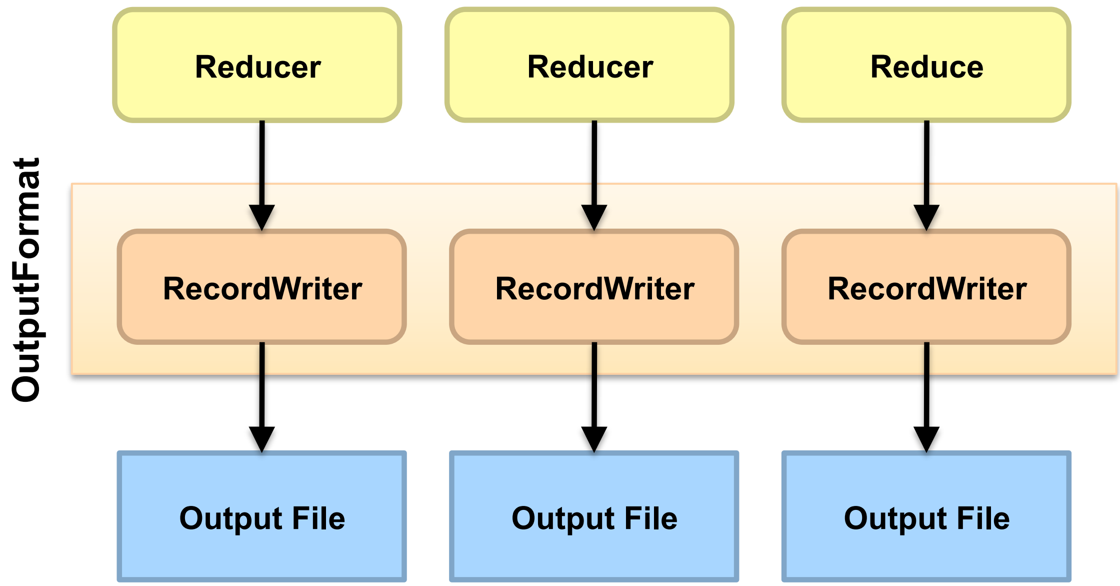
- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt
  - Multiple jobs can be composed into a workflow
- Job submission process
  - Client (i.e., driver program) creates a job, configures it, and submits it to job tracker
  - JobClient computes input splits (on client end)
  - Job data (jar, configuration XML) are sent to JobTracker
  - JobTracker puts job data in shared location, enqueues tasks
  - TaskTrackers poll for tasks
  - Off to the races…




Source: redrawn from a slide by Cloduera, cc-licensed




Source: redrawn from a slide by Cloduera, cc-licensed




Source: redrawn from a slide by Cloduera, cc-licensed

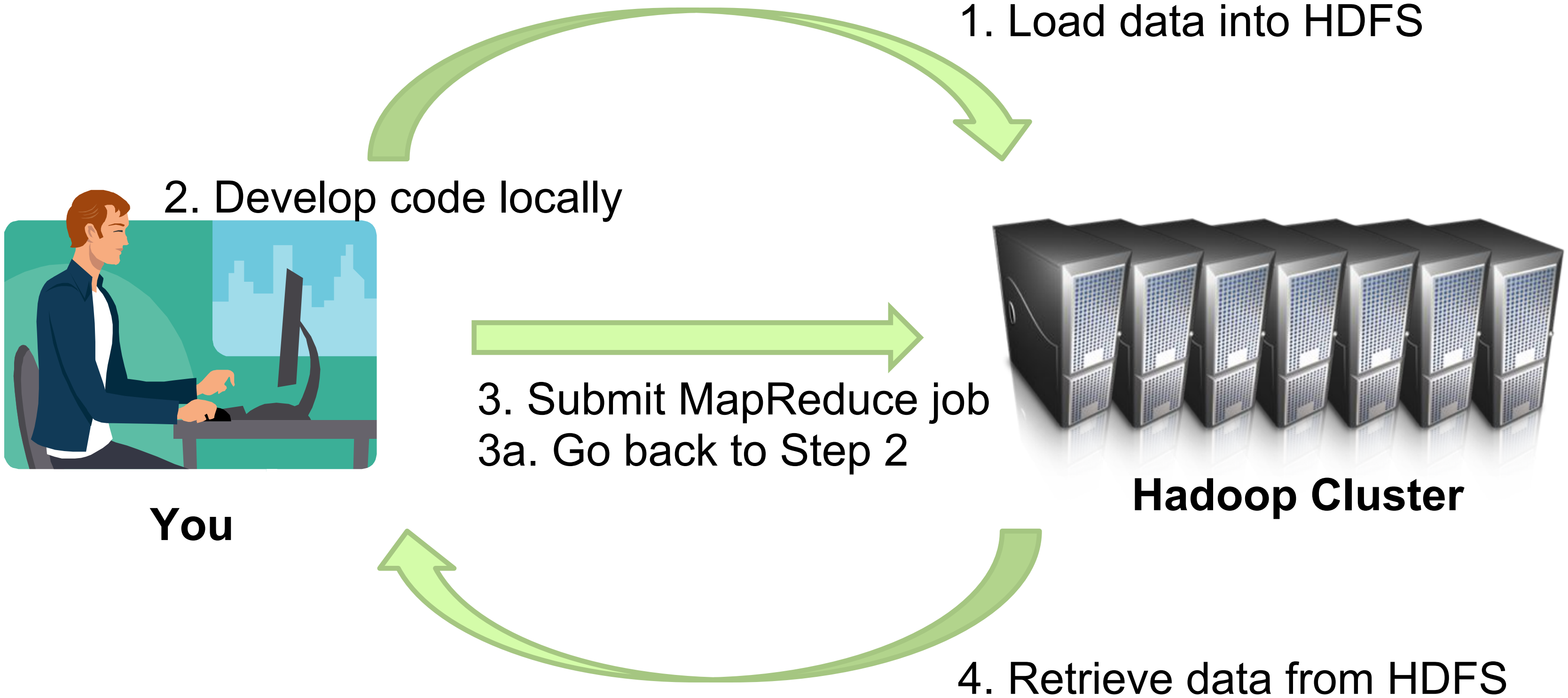
# Input and Output

- InputFormat:
  - TextInputFormat
  - KeyValueTextInputFormat
  - SequenceFileInputFormat
  - …
- OutputFormat:
  - TextOutputFormat
  - SequenceFileOutputFormat
  - …

# Shuffle and Sort in Hadoop

- Probably the most complex aspect of MapReduce!
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are "spilled" to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs here
- Reduce side
  - First, map outputs are copied over to reducer machine
  - "Sort" is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs here
  - Final merge pass goes directly into reducer

# Hadoop Workflow



# Debugging Hadoop

- First, take a deep breath
- Start small, start locally
- Unit tests
- Strategies
  - Learn to use the webapp
  - Where does println go?
  - Don’t use println, use logging
  - Throw RuntimeExceptions

# Start Small, Local

- Many mappers can be written as an Iterable
- Test the iterator locally on known input to make sure the right intermediates are generated
- Double check using an identity reducer (again, locally)
- Test reducer locally againts Iterable output
- Run on cluster on moderate data, debug again

# Unit Tests

- Whole courses / books on test-driven design
- Basic Idea
  - Write tests of what you expect the code will produce
  - Unit test frameworks (like JUnit) run those tests for you
  - These tests should always pass!  (Eclipse can force you)
- Write tests ASAP
  - Catch problems early
  - Ensure tests **fail**
  - Modular design to your code (good for many reasons)
- Write new tests for every bug discovered
- Only Jeff Dean, Chuck Norris, and Brian Kernighan write perfect code

# Unit Test Example (HW 2)

```
@Before
```

```
@Test
public void testOneWord() {
    List<Pair<PairOfStrings, FloatWritable>> out = null;

    try {
        out = driver.withInput(new LongWritable(0), new Text("evil::mal")).run();
    } catch (IOException ioe) {
        fail();
    }

    List<Pair<PairOfStrings, FloatWritable>> expected =
                        new ArrayList<Pair<PairOfStrings, FloatWritable>>();
    expected.add(new Pair<PairOfStrings, FloatWritable>
                        (new PairOfStrings("evil", "mal"), EXPECTED_COUNT));
    expected.add(new Pair<PairOfStrings, FloatWritable>
                        (new PairOfStrings("evil", "*"), EXPECTED_COUNT));

    assertListEquals(expected, out);
}
```

**Send input to mapper**

**Precompute the expected output**

**Check that they were actually the same**

# Recap

- Hadoop data types
- Anatomy of a Hadoop job
- Hadoop jobs, end to end
- Software development workflow


Source: Wikipedia (Japanese rock garden)