CS 378 – Big Data Programming

Lecture 9

Complex “Writable” Types

Review

- Assignment 4 - CustomWritable

- Questions/issues?

Hadoop Provided Writables

- We’ve used several Hadoop Writable classes
  - `Text`
  - `LongWritable`
  - `ArrayWritable`
    - Extended as `LongArrayWritable, DoubleArrayWritable`
- Hadoop provides many other classes
  - Wrappers for all Java primitive types
  - Some for Hadoop usage (`TaskTrackerStatus`)
  - Others for us to extend (`MapWritable`)

User Defined Writables

- Hadoop provided classes cover commonly used types and data structures
- But we're likely to need more application specific data structures/types
  - For example, `WordStatistics`
- We can define these one by one
  - Must implement the `Writable` interface
  - This will become tedious

User Defined Data Types

- Where might we look for a solution?
  - How are *ad hoc* types transferred elsewhere?
- Web formats for data structures
  - XML, JSON
  - Plus: Human readable, self describing
  - Minus: verbose, serialization is slower
- Java serialization
  - We have to write the serialization code
  - Again tedious, as data types get complex

User Defined Data Types

- RPC mechanisms
  - Marshall data in objects to be transferred to a "remote" procedure (no shared memory)
  - Usually procedure calls share memory
- Java serialization is one such mechanism
- Some others we'll look at:
  - Google protocol buffers (protobufs)
  - AVRO

Protobuf and AVRO

- These two approaches are interesting in that
  - They allow us to define complex types via a schema or IDL (**I**nterface **D**efinition **L**anguage)
  - They handle all the data marshalling/serialization
  - They create ”bindings” for various langauges
- AVRO was designed for use with Hadoop
- Protobufs require a `Writable` wrapper
  - May be provided now, wasn’t a few years ago

Protobuf Basics

- Protocol buffers (protobufs) used extensively at Google as the RPC mechanism
  - Multiple language support (Java, C++, Python)
  - Used in the Google map-reduce framework

- The schema language (IDL) defines “messages”
  - Data structures containing primitive data types
  - Required or optional
  - Repeated (array)
  - Embedded message

Protobuf Example

```
package  stats;
option java_package = “com.refactorlabs.cs378.utils”;
option java_outer_classname = “WordStatisticsProto”;

message WordStatistics {
  required int64 document_count = 1;
  required int64 total_count = 2;
  required int64 sum_of_squares = 3;
  optional double mean = 4;
  optional double variance = 5;
}
```

Protobuf Basics

- Protobuf fields
  - Scalars
  - Enumerations
  - Local message types
- Fields can be required or optional
  - Required field will always be present (and take up space)
  - Optional fields take no space when they have no value
- Fields can be repeated
- Fields can have a default value

Protobuf Basics

- With a protobuf defined, we “compile” it to create “bindings” to a language
- Output is Java source code
  - Package and class name as we defined them

- So what does this Java class do for us?
  - Allows instance to be created and populated
  - Allows access to the data stored therein
  - Performs serialization
    - This is one main reason for using protobufs
    - We’ll need to wrap this in a `Writable` to use it in mapReduce

Protobuf Generated Code

- Accessors for the internal data
  - Has methods
    - `hasDocumentCount()`
    - `hasTotalCount()`
    - …
  - Get methods
    - `getDocumentCount()`
    - `getTotalCount()`
- Builder class for constructing instances
  - Above methods
  - Plus set and clear methods

Protobuf Generated Code

- Repeated fields have some extra methods
  - `count()` method
  - Get and Set methods that take an index
  - `add()` method
  - `addAll()` method
- Instances are constructed with the `Builder` class
- Once created, the instance is immutable
- Enums and embedded message types become nested enums or classes

Protobuf I/O

- Protobufs do serialization via these methods
  - `writeTo(OutputStream out)`
  - `parseFrom(InputStream in)`

- To make it `Writable`, we can wrap the protobuf object with a class that:
  - Adapts `write()` to `writeTo()`, and
  - Adapts `readFields()` to `parseFrom()`

- Example:

Schema Evolution

- As your data changes and you update the message definition
- Old Java code can read and use data written under the new schema
  - It simply doesn't see the new fields
- New Java code can read and use data written under the old schema
  - New fields added must be optional
  - The has() methods can be used to determine where new fields are unpopulated

Other Protobuf Benefits

- Efficient in terms of space
  - Optional fields with no value – not in the output
  - Data values compressed

- Efficient in terms of speed
  - Object construction from input is fast
  - Object contents to output is fast

- But AVRO is more widely used, so we'll examine it in the next class

Protobuf Basics

- More info one protocol buffers can be found here:
  - https://developers.google.com/protocol-buffers/docs/javatutorial
  - http://talks.spline.de/slides/protobuf.pdf

- Note: These references do not address using protobufs with Hadoop map-reduce