Apache Avro

Sharing and managing data efficiently

Scott Carey
Apache Avro PMC Chair
Principal Architect, RichRelevance
scottcarey@apache.org

March 6, 2012

What is Apache Avro?

"Apache Avro™ is a data serialization system."

(avro.apache.org)

- Binary and JSON serialization
- File Format
- HTTP and raw socket RPC
- Schema evolution and management

Motivation

- Protocol Buffers and Thrift already exist and work well.
- Compact and efficient
- Expressive Schemas
- Code generation is required.
  - Unable to browse arbitrary data

The Avro Serialization Approach

The Avro schema used to write data is required to be available when reading it.

- Fields are not tagged
  - More compact
  - Potentially faster
- Code generation is optional.
  - Simple implementations can read and write data
  - Dynamic, discoverable RPC is also possible (but not implemented)
- Schema storage explicitly or by reference required.

Compactness

```
class Card {
  int number; //ace = 1, king = 13
  Suit suit;
}
enum Suit {
  SPADE, HEART, DIAMOND, CLUB;
}
```

Java Heap: 24 bytes (32 bit JVM) to 32 bytes
Avro binary: 2 bytes

Compactness

```
Card card = new Card();
card.number = 1;
card.suit = Suit.SPADE;
```

Avro binary: `0x02 0x00`

First byte: the integer 1, encoded

Second byte: the ordinal of the Suit enum (0), encoded

Compactness Notes

Without a Schema, the binary is ambiguous

- Fields are not tagged or delimited
- Nested records are not tagged or delimited

Languages

Avro Implementations in Apache Avro

- Java, C, C++, C#, php, python, ruby

Other implementations outside of Apache exist in some languages

- Perl

The Center of the Avro Universe: Avro Schemas

- A Schema is JSON text
    - Language implementations do not need a custom parser
    - Human readable, but not extremely friendly
- AvroIDL is a human friendly Schema definition language
    - The Java implementation and build tools convert this to JSON
    - Other language implementations need only know the core JSON schema specification.

Avro Schema Evolution

A reader always has a schema pair:

- The schema that the data was written with
- The schema that the reader wishes to interpret the data as
- These may be the same

Schema read resolution identifies components by ***name***

- Extra fields are skipped
- Missing fields are populated with defaults
- Name aliases are supported

Simple Avro Schema

JSON

```
{"name":"Coffee", "type":"record":,
 "fields": [
  {"name":"brand", "type":"string"},
  {"name":"ounces", "type":"float"},
  {"name":"caffeinated", "type":"boolean"}
 ]
}
```

AvroIDL

```
record Coffee {
  string brand;
  float ounces;
  boolean caffeinated;
}
```

Defaults and Nullable Fields

JSON

```
{"name":"Coffee", "type":"record":,
 "fields": [
  {"name":"brand", "type":["null","string"], "default":null},
  {"name":"ounces", "type":"float"},
  {"name":"caffeinated", "type":"boolean", "default":true}
 ]
}
```

AvroIDL

```
record Coffee {
 union { null, string } brand = null;
 float ounces;
 boolean caffeinated = true;
}
```

Schema Evolution

Old

```
record Coffee {
  float ounces;
  boolean caffeinated;
}
```

New

```
record Coffee {
  union { null, string } brand =
null;
  float ounces;
  boolean caffeinated = true;
}
```

- Data serialized with the old definition can be read with the new.
- The brand field will be materialized with the default, value.
- No manual version management in client code!

Recursive Schema

JSON

```
{"name":"Tree", "type":"record":,
 "fields": [
  {"name":"value", "type":"long"},
  {"name":"left", "type":["null","Tree"]},
  {"name":"right", "type":["null","Tree"]}
}
```

AvroIDL

```
record Tree {
  long value;
  union { null, Tree } left;
  union { null, Tree } right;
}
```

Expressive Schemas

8 primitive types

```
null, int, long, float, double, string, bytes, boolean
```

6 complex types

```
record, array, map, union, fixed, enum
```

An Avro schema can represent any non-recursive in memory data structure.

This is the same subset of data structures that can be made immutable.

Avro File Format

Avro defines a file format optimized for storing bulk data.

- Each File contains records written with one schema
- Records are packed into blocks of configurable size
- Blocks are optionally compressed
- File header contains schema, extensible meta-data
- Good for Hadoop MapReduce bulk processing and archival storage
    - Works well with schema evolution
    - New and old data can be viewed with new or old versions of a schema from new or old code

Avro Java API

Map Objects to Schemas

- Specific API -- Compile Schemas into Java classes
- Generic API -- Represent data from any Schema without code generation
- Reflect API -- Map existing classes to Avro Schemas

Avro Specific Java API

```
src/main/avro/coffee.avsc

{"name":"Coffee", "type":"record":,
 "fields": [
  {"name":"brand", "type":"string"},
  {"name":"ounces", "type":"float"},
  {"name":"caffeinated", "type":"boolean"}
 ]
}
```

Avro Specific Java API

```
pom.xml
 <plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <executions>
   <execution>
    <goals>
     <goal>schema</goal>
    </goals>
    <configuration>
     <sourceDirectory>src/main/avro</sourceDirectory>
     <outputDirectory>target/generated-sources/avro</outputDirectory>
    </configuration>
   </execution>
  </executions>
 </plugin>
```

Avro Specific Java API

```
// Coffee class generated by the bulid system from schema
Coffee coffee = Coffee.newBuilder()
  .setBrand("excellent")
  .setCaffeinated(true)
  .setOunces(7.5f)
  // validates that object is valid
  .build();


DatumWriter writer = new SpecificDatumWriter(Coffee.$SCHEMA);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
BinaryEncoder encoder = EncoderFactory.get().binaryEncoder(baos,
null);
writer.write(coffee, encoder);
encoder.flush();
```

Avro Java Generic API

```
// get by parsing from JSON or by creating with Schema API
Schema coffeeSchema = ...
GenericRecord coffee = GenericRecordBuilder(coffeeSchema)
  .set("brand","excellent")
  .set("caffeinated", true)
  .set("ounces",7.5f)
  // validates that object conforms to schema
  .build();

DatumWriter writer = new GenericDatumWriter(coffeeSchema);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
BinaryEncoder encoder = EncoderFactory.get().binaryEncoder(baos,
null);
writer.write(coffee, encoder);
encoder.flush();
```

Avro Java Reflect API

```
// Pre-existing Coffee class
Coffee coffee = ...

// Attempt to induce a schema from the class reflectively
DatumWriter writer = new ReflectDatumWriter(Coffee.class);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
BinaryEncoder encoder = EncoderFactory.get().binaryEncoder(baos,
null);
writer.write(coffee, encoder);
encoder.flush();
```

Reflect API has a limited set of field types and access it supports. Contributions for enhancement welcome!

More Avro Java APIs

Low Level APIs

- {Decoder,Encoder} -- translate Avro primitives to binary or JSON
- Validating{Encoder,Decoder} -- validate that a stream of primitives corresponds to an Avro Schema
- ResolvingDecoder -- translate data written with one schema version to appear as another when read, if compatible

File Writing/Reading via org.apache.avro.file package

MapReduce Related Frameworks

Pig, Hive, Crunch, HCatalog, Sqoop, Flume, Cascading

- Various levels of Avro support.

HBase -- HAvroBase, others. Managing types in HBase using Avro

https://github.com/bixolabs/cascading.avro

http://www.cloudera.com/blog/2011/07/avro-data-interop/

Avro MapReduce API

Single Valued Inputs and Outputs, Key/Value pairs only for intermediate

```
map(IN, Collector<OUT>)  -- map only job
map(IN, Collector<Pair<K,V>>)
reduce(K, Iterable<V>, Collector<OUT>)
```

Community enhancements / extension under review, see
https://issues.apache.org/jira/browse/AVRO-593

Avro MapReduce Example

```
Schema KEY_SCHEMA = Schema.create(Type.BOOLEAN);
Schema VAL_SCHEMA = Schema.create(Type.FLOAT);
public static class MapImpl
    extends AvroMapper<Coffee, Pair<Boolean, Float>> {
  public void map(Coffee c,
      AvroCollector<Pair<Boolean, Float>> collector,
      Reporter reporter) {
    collector.collect(
      new Pair<Boolean, Float>(
        c.getCaffeinated(), KEY_SCHEMA,
        c.getOunces(), VAL_SCHEMA));
    }
  }
```

Avro MapReduce Example

```
public static class ReduceImpl
    extends AvroReducer<Boolean, Float, String> {
  public void reduce(Boolean caffeinated, Iterable<Float> drinks,
      AvroCollector<String> collector, Reporter reporter) {
    double howMuch = 0.0d;
    for (Float size : drinks) {
      howMuch += size.doubleValue();
    }
    String caff = caffeinated ? "regular" : "decaf";
    collector.collect("Imbibed " + howMuch +
      " ounces of " + caff + " coffee.");
  }
}
```

Avro RPC

The Avro Spec defines

- An HTTP protocol
  - Most compatible
  - Good performance
  - In-order responses only
- A raw socket protocol using optional SASL authentication
  - High performance
  - SASL has virtually no overhead when security not enabled
  - One-way messages
  - Asynchronous, out-of-order RPC under discussion for possible addition

Avro RPC

Netty Protocol in Java

- Asynchronous API, requests, responses
- Not in spec (yet?)

Other Avro Features

- Thrift and Protocol Buffer adapters. Serialize preexisting data beans to Avro binary.
- Code Generation plug-ins. Use a custom pattern or generate for a new language.
- Blocked array encoding. Serialize an array with length not known at the time writing starts.

A few Avro Ideas

- Extracting schema subsets, 'AvroPath'
- An API more tolerant of truncated / corrupted streams.
- Column oriented encoding
- Enhanced Java Reflection API, perhaps leverage bytecode generation and "Mix-In" annotations.
- More languages! More features in each!
- Schema builder API for Java

Questions?