Pipeclamp

Intent

While Avro extends JSON by incorporating schemas to specify record layouts, it doesn't provide support for field specifications beyond their core datatypes.   For example a numeric field can denote a person's height or weight but nothing more. What we'd like is to include some additional sanity checks: positive numbers only, some realistic upper bounds. Maybe even something that adjusts the bounds based on age. Being able to declare just what these limits are would go a long way to catching errors early and improve the confidence we have in the system.

How it works

Pipeclamp extends the Avro schemas by utilizing additional keywords that allow developers specify field and record-level constraints. At runtime, a validation tool can be used to enforce them and identify any records whose field values lie outside the specified limits. In streaming environments, a related tool can also be used to collect live metrics on fields of interest.

<positioning examples>

The new Pipeclamp keywords in the schema do not interfere with serialization or other Avro-related functionality, they are just ignored. The records with the new constraints do not change either, the changes are just limited to the schemas.

Applications

In practice, Pipeclamp constraints are declarative, they do not provide enforcement behavior until they are wired into contexts where their evaluations can be acted upon.
If Avro plays a critical role in your enterprise, Pipeclamp can be positioned as an effective chokepoint or firewall to block the transmission of faulty records closer to the source and avoid causing angst among downstream dependents.

Within the UI

One of the challenges with building effective UIs over the long term is ensuring that rules coded in place on the front end remain in sync with all restrictions in effect downstream of it. If a text entry field on a form is set to allow a maximum of 30 alphanumeric characters, chances are good that this max length is mirrored by a field in a database table and/or some output field in a report. Updating any one of these separate systems doesn't ensure that the remaining ones stay in sync.
<finish with example>

Streaming environments

Successful transmission of live data between teams or customers relies on the use of specifications easily available to everyone. If these are set forth in an email or

PowerPoint slide chances are high they won't be updated when any of the schemas are. Chaos ensues.
While it would be nice if Pipeline could validate constraints in effect in databases (DDL checks), this isn't practical nor guaranteed to catch restrictions hidden in associated code.

<show enterprise outline>

Constraints

Objects, fields, and collections in a schema can include one or more constraints by adding the new 'constraints' keyword to the related node:

```
fields : {
        "name": "firstname",
        "type": "string",
        "doc": "Person's first name",
        "constraints" : [
                {
                "function" : "length",              // pre-registered
                "id" : "firstNameLength",           // must be unique
                "args" : [
                        { "name" : "min", "value" : "2"},
                        { "name" : "max", "value" : "30"}
                        ]
                } ]
},
```

In this addition to a partial schema, the function refers to one of several registered for string fields while the id tag identifies it among any other constraints that might exist on the same field. If and when an out-of-bounds condition is detected, the id value will be included in the violation for traceability.
Most constraints include one or more parameters and are specific to the exact constraint function.

When the Pipeclamp validator scans a schema for constraints, basic sanity checks are performed to ensure they are fully-formed and uniquely identified.

The validator

At runtime, a validator can be configured manually by registering constraints against specified field paths in a manner similar to XPath:

```
validator = new AvroValidator(ErrorEvent.getClassSchema(), false);
validator.register(
        new StringLengthConstraint("partnerLength", false, 3, 30),
        new SimpleAvroPath<String>("header", "partner")
        );
```

```
validator.register(
        new RegexConstraint(RegexDescriptor.Hostname),
        new SimpleAvroPath<String>("header", "hostname")
        );
```

Its more likely setup would be taken from the schema files directly.

Once configured for a specific record type, the validator can be given instances of that type and have it report on any violations found:

```
Person p = new Person(……);
Map<Path, Collection<Violation>> issues = validator.validate(p);

if (issues.isEmpty()) return;
        // else invoke issuehandler..
for (Entry<Path,Collection<Violation>> errors : issues.entrySet()) {
        System.err.println(……);
        }
```

Developers can then take action based on the results returned by either filtering the failures from their application and/or logging it to some relevant monitor.

The default validator, when using the existing constraints, is stateless and can be safely shared across multiple threads.

Available constraints

The following type-specific constraints are currently provided:

String

Length - min/max

Probably one of the more useful and easiest-to-understand constraints that allows you to specify a minimum and/or maximum required string value length. Can be used to inhibit empty values or nonsensical things like last names composed of single characters. Does not trim whitespace before evaluation.

IllegalChars

Allows one to specify the set of illegal characters not to be found in a string value. Violations from this constraint will identify the first illegal character found.

Whitespace

Checks for unwanted leading or trailing whitespace characters.

NumericOnly

Digits only with some optional allowable characters. I.e. “ 604 987 2520”

Words – required/restricted

Can be used to identify phrases containing words we expect to find (or not) in a longer text field. These could include common swear words or typos.

This constraint is more of a demonstrator, a more useful version might make use of larger cache backed by an external database.

Regex

The most powerful constraint in the arsenal but also the most expensive to run. We would suggest trying the other constraints in combination before applying this one if performance is an issue. Also, violations issued by a single regex constraint might not be as descriptive as having several simpler ones.

NamedRegex

Pipeclamp includes several common regex expressions that can be referenced by name instead of having to spell them out explicitly. Not only does this prevent subtle errors but ensures that any improvements made to them are applied across the system.
Currently supported regexes include: 'ipAddress', 'emailAddress', 'hostname', 'zipcode', 'IPV4', 'IPV5', and 'url'.

Numeric

Value – min/max

Allows you to specify a minimum and/or maximum value for any numeric datatype. Chances are, most of the numeric fields you have should not include negative numbers so this constraint is just what you need to ensure you don't get any.

NamedRange

Like the NamedRegex constraint, the NamedRange has a few pre-defined ranges for common numeric fields: latitude (-180 to +180), longitude (-90 to +90)

Histogram (integers only)

Useful the field is limited to a limited number of integer values.

Timestamp

Range

Allows you to specify a legal time range with absolute dates or as offsets from the current date.

Era – before/after

Collections

Items – min/max

Arrays or maps can be required to have a minimum and/or maximum number of entries.

Contents – noneOf, oneOf, allUnique, etc

Ensures that the collection abides by the stated restriction

Classifications – mustBe, mustNotBe
Uses a classification function to filter in/out collection elements

UniqueField
Allows you to designate a field as holding unique values across all instances to avoid duplicate references. For example, an array holding vehicles might designate a VIN field.
If the constraint is applied to a map collection this could be redundant.

Enum

RestrictedItem
Supports the use of deprecation for older options where older choices might not be permitted anymore.

ByteArray

Length – min/max
Just like the string length constraint, ensures properly sized byte array values.

Prefix
For fields holding binary data like images, sound files, or archives, you can limit them to filetypes with known prefix identifiers.   See https://en.wikipedia.org/wiki/List_of_file_signatures

Record

Presence – null/not-null
Useful if the Avro schema permits null values (union with null) while your application doesn't.

MaxDepth
For recursive structures

Custom constraints
All the major types in Pipeclamp are represented by interfaces so you can create your own constraint and register it in the factory before any schemas are scanned.

example

Just be aware of any thread safety or performance issues if you implement constraints that call upon external data sources.

Metrics

The metrics subsystem works in a similar manner to the constraints however, instead of returning possible violations after retrieving a field value, it can be used to provide statistics over them for later analysis.

If fields on schema include metrics tags, an aggregator is used to collect values from the designated fields. When the end of a sampling period is reached, summary values can be extracted upon which a new sampling interval is initiated.

Specifying metrics within the schema itself might not be appropriate if use cases for the data spans multiple consumers. In these cases, metrics could be specified manually depending on the interests of each consumer.

That said, the following example shows how we're interested in determining the average height of people across a group:

```
fields : {
      "name": "height",
      "type": "int",
      "doc": "person's height in inches",
      "default": 1,
      "metrics" : [
            { "function" : "average", "id" : ”personHeightAvg" }
             ]
      },
```

Like the constraints, one can specify more than one metric per field as long as they have unique identifiers.

Providing the same metric programmatically:

<show with descriptor>

Available metrics

Record

count

Numeric

Count, Sum

Min-max

Histogram

String

Min-max length

Average length

Timestamp

Histogram

Collections

Min-max size

Histogram (item type)

Performance impact

The impact of the metrics system is proportional to the number and type of metrics configured along with the sampling frequency. Some metric types (i.e. avg, std deviation) need to retain every value within a sampling period before they can compute a result. Others (min, max) would just retain boundary values. Collecting the samples over a period incurs negligible cost aside from metric-specific memory consumption. Computing the metrics at the end of a period *<TODO>*

Metric	Memory	Cpu		
sum, count	Low	Very low		
min-max	Low, 2 values per field	Very low		
average	High, all values collected	low		
histogram	Low-medium	low		

…more…

The aggregator

Like the validator, the default aggregator can pull metric specifications from a schema or can accept them programmatically. It needs to know the duration of the sampling period along with a callback function to invoke with the values after the metrics are computed.

Custom metrics

All major Pipeclamp elements are characterized by interfaces so providing your own aggregator is fairly straightforward: just develop them and ensure they get registered with the aggregator at runtime.

Find below a near-complete example showing the inner workings of the Averager metric:

```
public class Averager<I extends Number> extends AbstractCollectionFunction<I,I> {

       private final Summer<I> summer;
       private final Multiply<I> multiplier;

public Collector<I> createCollector() { return new FullCollector<I>(predicate()); }

public I compute(Collector<I> collector) {
       I sum = summer.compute(collector);
       return multiplier.divide(sum, collector.collected());
       }
```

Unlike the String metrics, which just work with a single datatype, numeric metrics should operate across all Number subtypes. These are accounted for via type-specific singletons declared in the class:

```
Averager<Integer>IntAvg = new Averager<Integer>(Summer.IntegerSum, Multiply.IntMult);
Averager<Long>    LongAvg = new Averager<Long>(Summer.LongSum, Multiply.LongMult);
Averager<Float>   FloatAvg = new Averager<Float>(Summer.FloatSum, Multiply.FloatMult);
Averager<Double>  DoubleAvg = new Averager<Double>(Summer.DoubleSum, Multiply.DoubleMult);
Averager<BigDecimal> BigDecimalAvg = new Averager<BigDecimal>(Summer.BigDecimalSum,
Multiply.BigDecimalMult);
```

(`public static final` *modifiers omitted for brevity)*

These are made retrievable by type through a common map:

```
Map<Class<?>, Function<?,?>> AveragersByType = ImmutableMap.<Class<?>,
Function<?,?>>builder().
                     put(Integer.class, IntAvg).
                     put(Long.class, LongAvg).
                     put(Float.class, FloatAvg).
                     put(Double.class, DoubleAvg).
                     put(BigDecimal.class, BigDecimalAvg).
                     build();
```

After completing your custom metric (and unit tests), ensure it gets used by the aggregator by registering it before any schemas that reference them are parsed:

Predicates
All metrics can be provided with a predicates that act as filters for incoming values. These can be used to reject:

- unwanted outliers not caught by any earlier constraints.
- records whose timestamps fall outside a specified window

Implementation details

Paths
Like XPath for XML, Pipeclamp uses a similar approach for navigating data structures. In order for the system to check or monitor a field value, you have to provide a means to get to it starting from the root element.

The simplest paths consist of string sequences that match the field names set out in the schema:

<example>

Paths that route to elements held in collections are a bit more complex:

<example>

Depending on field and the constraint/metric being used, Pipeclamp selects between several Path implementation types:

SimpleAvroPath
Uses simple string sequence to descend through the object and retrieve values or collections.

JsonPath
A wrapper that makes use of the JayWay project. Provides support for complex expressions, useful for relational constraints that span multiple fields.

ReflectivePath
If you'd like to apply Pipeclamp to simple POJOs you could use the ReflectivePath. It uses a sequence of method names to retrieve the values of interest: