Nested Fields Selection Support

By Chris K Wensel<chris@wensel.net>; first draft 10-19-16

This document is intended to capture feedback for an extension to Cascading that would allow for selection operations into Tuples to extend past the Tuple and into the managed object.

Cascading relies on the concept of a Tuple to manage data hand off from input data-sets through internal operations to outgoing data-sets.

A Tuple is essentially a List or Array of values, addressable via in ordinal or index. Index 0 retrieves the first value in the Tuple. Where the values can be any primitive value or complex object. That is, the first position in a Tuple can hold an Integer, and the second position (index 1) can hold an arbitrary domain object like Person.

To help people manage complexity, values in Tuples can be addressed via field names, by declaring the fields in a given Tuple with the Fields and TupleEntry classes.

```
TupleEntry entry = new TupleEntry( new Fields( "c", "b" ), new Tuple( "C", "B" ) );
assert "C".equals( entry.getObject( "c" ) );
```

Fields are used to declare the the contents of a Tuple, they may optionally include type information along with the field names to help the user and the Cascading planner.

Fields are also used to select values from a Tuple into a new Tuple.

```
TupleEntry entry = new TupleEntry( new Fields( "c", "b" ), new Tuple( "C", "B" ) );
assert "C".equals( entry.selectTuple( new Fields( "c" ) ).get( 0 ) );
```

The above code returns a new Tuple instance with the values selected by the field names passed to the #selectTuple() method.

The flat Tuple model is sufficient for many applications. This is especially true when reading a database or flat text files like CSV or TSV where each line is a record, and each record becomes a Tuple.

When the data is hierarchical, as it is with JSON, the flat model can be cumbersome.

Scope

Cascading provides a useful field algebra that allows for efficient manipulation of Tuples with little to no object creation and garbage-collection, because the manipulations are declared by the developer up-front.

This proposal focuses on the user-facing API and semantics that extend the field algebra and upfront declaration to allow for efficient use of nested objects and hierarchical data structures.

Not in scope for this proposal is in-place modification or transformation of a nested Object.

Tuple instances passed downstream to user defined Operations are marked as immutable as a safety measure and to improve runtime performance. If a user defined Operation could modify a Tuple directly that was created up-stream, object instance re-use will be compromised.

In addition, if Tuples were mutable, the efficiency provided by up upfront declaration of field selection would be lost. This efficiency in part if gained by presenting updateable 'views' across multiple up-stream Tuple instances and/or views, instead of creating a new argument Tuple for every operation invocation.

That said, this has always been a gray area in Cascading in respect to Tuple immutability and contained object modification and should be addressed in the future with a clear contract or additional functionality (runtime change protection and possible deep copy operations on nested object types).

Design

Goals

- Upfront expression validation during planning; if Person does not declare 'age', selecting 'father.age' should fail early in the same way as a missing field.
- Ship with support for domain Objects
- Nested type extensibility to allow JSON or other models
- Expression language extensibility
- Little or no performance loss
- Allow for lazy evaluation/deserialization of nested hierarchical data

Not in Scope

- In-place transformation of nested objects
- One-size-fits-all expression language -- expressions likely tied to nested object type

Behavior

Is it expression implementation specific if the expression should return ‘null’ or fail if expression addresses unavailable fields/objects.

E.g, the expression “father.name.first” should return ‘null’ if ‘name’ is null on the ‘father’ Person instance, or should it fail.

API Proposal

When declaring a new Each or Every operation, an argument selector is provided by the developer to declare what arguments should be passed to the managed operation.

```
new Each( previous, new Fields( “first”, “last” ), new ConcatFunction( new Fields( “name” ) );
```

The above example selects the ‘first’ and ‘last’ fields from the stream and passes them to the ConcatFunction operation in order to create a new field in the stream named ‘name’.

The following sections describe the API to declare a Tuple with nested fields, how a field selector would retrieve a result (arguments to an operation) Tuple that declares one or more nested fields, and finally, how an operation could access values inside a nested field or hierarchical data structure.

Note class names are subject to change, the API signatures are what’s important.

Fields Declaration

```
new NestedFields( "father" ); // will behave as a normal Fields instance

// next two are equivalent
new NestedFields( "father.age" );
// allows for Fields instance re-use - 'father' could ba a constant
new NestedFields( new Fields( "father" ), "age" );

// object supported by default - wraps Person class with ObjectNestedType instance
new NestedFields( Person.class, "father.age" );
new NestedFields( Person.class, "father.doesNotExist" ); // fails

// allows for Fields instance re-use - 'father' could ba a constant or type inherited
new NestedFields( new Fields( "father", Person.class ), "age" );
new NestedFields( new Fields( "father", Person.class ), "doesNotExist" ); // fails
```

```
// can declare a NestedType expression handler -- or annotation on Person class
NestedType nestedType = new ObjectNestedType( Person.class );
new NestedFields( new Fields( "father", nestedType ), "age" );
new NestedFields( new Fields( "father", nestedType ), "doesNotExist" ); // fails

// must be able to build a selector
new NestedFields( new Fields( "father" ), "name" )
  .append( new NestedFields( new Fields( "mother" ), "name" ) );

// retrieves the 'age' value, but the field is named 'fathersAge'
new NestedFields( new Fields( "father", nestedType ), "age" )
  .as( "fathersAge" );
```

Operation Argument Selection

```
tuple = new Tuple();

firstPerson = new Person( "john", "doe", 40 );
secondPerson = new Person( "jane", "doe", 39 );

tuple.add( firstPerson );
tuple.add( secondPerson );

fields = new Fields( "father", Person.class )
  .append( new Fields( "mother", Person.class ) );

entry = new TupleEntry( fields, tuple );

assertEquals( new Tuple( firstPerson ), entry.selectTuple( new Fields( "father" ) ) );
assertEquals( new Tuple( secondPerson ), entry.selectTuple( new Fields( "mother" ) ) );

assertEquals( new Tuple( secondPerson, firstPerson ),
  entry.selectTuple( new Fields( "mother", "father" ) ) ); // order preserved

assertEquals( new Tuple( "john" ), entry.selectTuple( new Fields( "father.name.first" ) ) );
assertEquals( new Tuple( "jane" ), entry.selectTuple( new Fields( "mother.name.first" ) ) );

assertEquals( new Tuple( firstPerson, 40, "john" ),
  entry.selectTuple( new Fields( "father", "father.age", "father.name.first" ) ) );
```

Value Retrieval via Expression

```
assertEquals( firstPerson, entry.getObject( new Fields( "father" ) ) );
assertEquals( secondPerson, entry.getObject( new Fields( "mother" ) ) );

assertEquals( firstPerson, entry.getObject( new NestedFields( "father" ) ) );
assertEquals( secondPerson, entry.getObject( new NestedFields( "mother" ) ) );

assertEquals( 40, entry.getObject( new NestedFields( "father.age" ) ) );
assertEquals( 39, entry.getObject( new NestedFields( "mother.age" ) ) );

assertEquals( 39, entry.getInteger( new NestedFields( "mother.age" ) ) );
assertEquals( "39",
  entry.getString( new NestedFields( "mother.age" ) ) ); // coercions supported

assertEquals( "john", entry.getObject( new NestedFields( "father.name.first" ) ) );
assertEquals( "jane", entry.getObject( new NestedFields( "mother.name.first" ) ) );

assertEquals( "john", entry.getObject( new NestedFields( "0.name.first" ) ) );
assertEquals( "jane", entry.getObject( new NestedFields( "1.name.first" ) ) );
```

Joins

TBD

Expression Syntax

Given an expression of the form “father.name.first”, “father” addresses the father instance of the an object, the remaining string, “name.first” is opaque in the sense that it is up to the NestedType implementation to declare the expression parser and handler.

The implication is that any expression will be split on the first “.” (dot) in the expression. Where the first part of the split is treated as a normal field name (if a number will be coerced into a number), the later part of the expression will be passed to the expression handler.

The default ObjectNestedType implementation will rely on the OGNL language syntax. OGNL allows for complex expressions than can include retrieving values from lists and maps.