Cascading 2.6 User Guide

Concurrent, Inc.

Publication date September 2014

Table of Contents

1. About Cascading

1.1 What is Cascading?

Cascading is a data processing API and processing query planner used for defining, sharing, and executing data-processing workflows on a single computing node or distributed computing cluster. On a single node, Cascading's "local mode" can be used to efficiently test code and process local files before being deployed on a cluster. On a distributed computing cluster using Apache Hadoop platform, Cascading adds an abstraction layer over the Hadoop API, greatly simplifying Hadoop application development, job creation, and job scheduling.

1.2 Usage Scenarios

Why use Cascading?

Cascading was developed to allow organizations to rapidly develop complex data processing applications with Hadoop. The need for Cascading is typically driven by one of two cases:

Increasing data size exceeds the processing capacity of a single computing system. In response, developers may adopt Apache Hadoop as the base computing infrastructure, but discover that developing useful applications on Hadoop is not trivial. Cascading eases the burden on these developers and allows them to rapidly create, refactor, test, and execute complex applications that scale linearly across a cluster of computers.

Increasing process complexity in data centers results in one-off data-processing applications sprawling haphazardly onto any available disk space or CPU. Apache Hadoop solves the problem with its Global Namespace file system, which provides a single reliable storage framework. In this scenario, Cascading eases the learning curve for developers as they convert their existing applications for execution on a Hadoop cluster for its reliability and scalability. In addition, it lets developers create reusable libraries and applications for use by analysts, who use them to extract data from the Hadoop file system.

Since Cascading's creation, a number of Domain Specific Languages (DSLs) have emerged as query languages that wrap the Cascading APIs, allowing developers and analysts to create ad-hoc queries for data mining and exploration. These DSLs coupled with Cascading local-mode allow users to rapidly query and analyze reasonably large datasets on their local systems before executing them at scale in a production environment. See the section on DSLs for references.

Who are the users?

Cascading users typically fall into three roles:

The application Executor is a person (e.g., a developer or analyst) or process (e.g., a cron job) that runs a data processing application on a given cluster. This is typically done via the command line, using a prepackaged Java Jar file compiled against the Apache Hadoop and Cascading libraries. The application may accept command-line parameters to customize it for a given execution, and generally outputs a data set to be exported from the Hadoop file system for some specific purpose.

The process Assembler is a person who assembles data processing workflows into unique applications. This work is generally a development task that involves chaining together operations to act on one or more input data sets, producing

one or more output data sets. This can be done with the raw Java Cascading API, or with a scripting language such as Scala, Clojure, Groovy, JRuby, or Jython (or by one of the DSLs implemented in these languages).

The operation Developer is a person who writes individual functions or operations (typically in Java) or reusable subassemblies that act on the data that passes through the data processing workflow. A simple example would be a parser that takes a string and converts it to an Integer. Operations are equivalent to Java functions in the sense that they take input arguments and return data. And they can execute at any granularity, from simply parsing a string to performing complex procedures on the argument data using third-party libraries.

All three roles can be filled by a developer, but because Cascading supports a clean separation of these responsibilities, some organizations may choose to use non-developers to run ad-hoc applications or build production processes on a Hadoop cluster.

1.3 What is Apache Hadoop?

From the Hadoop website, it “is a software platform that lets one easily write and run applications that process vast amounts of data”. Hadoop does this by providing a storage layer that holds vast amounts of data, and an execution layer that runs an application in parallel across the cluster, using coordinated subsets of the stored data.

The storage layer, called the Hadoop File System (HDFS), looks like a single storage volume that has been optimized for many concurrent serialized reads of large data files - where "large" might be measured in gigabytes or petabytes. However, it does have limitations. For example, random access to the data is not really possible in an efficient manner. And Hadoop only supports a single writer for output. But this limit helps make Hadoop very performant and reliable, in part because it allows for the data to be replicated across the cluster, reducing the chance of data loss.

The execution layer, called MapReduce, relies on a divide-and-conquer strategy to manage massive data sets and computing processes. Explaining MapReduce is beyond the scope of this document, but its complexity, and the difficulty of creating real-world applications against it, are the chief driving force behind the creation of Cascading.

Hadoop, according to its documentation, can be configured to run in three modes: standalone mode (i.e., on the local computer, useful for testing and debugging in an IDE), pseudo-distributed mode (i.e., on an emulated "cluster" of one computer, not useful for much), and fully-distributed mode (on a full cluster, for staging or production purposes). The pseudo-distributed mode does not add value for most purposes, and will not be discussed further. Cascading itself can run locally or on the Hadoop platform, where Hadoop itself may be in standalone or distributed mode. The primary difference between these two platforms, local or Hadoop, is that, when Cascading is running in local mode, it makes no use of Hadoop APIs and performs all of its work in memory, allowing it to be very fast - but consequently not as robust or scalable as when it is running on the Hadoop platform.

Apache Hadoop is an Open Source Apache project and is freely available. It can be downloaded from the Hadoop website: http://hadoop.apache.org/core/

1.4 Hadoop 1 vs Hadoop 2

Cascading 2.6 supports both Hadoop 1.x and 2.x by providing two Java dependencies, `cascading-hadoop.jar` and `cascading-hadoop2-mr1.jar`. These dependencies can be interchanged but the `hadoop2-mr1.jar` introduces new and deprecates older API calls where appropriate. It should be pointed out `hadoop1-mr1.jar` only

supports MapReduce 1 API conventions. With this naming scheme new API conventions can be introduced without risk of naming collisions on dependencies.

2. Diving In

The most common example presented to new Hadoop (and MapReduce) developers is an application that counts words. It is the Hadoop equivalent to a "Hello World" application.

In the word-counting application, a document is parsed into individual words and the frequency of each word is counted. In the last paragraph, for example, "is" appears twice and "equivalent" appears once.

The following code example uses Cascading to read each line of text from our document file, parse it into words, then count the number of times each word appears.

```
// define source and sink Taps.
Scheme sourceScheme = new TextLine( new Fields( "line" ) );
Tap source = new Hfs( sourceScheme, inputPath );

Scheme sinkScheme = new TextDelimited( new Fields( "word", "count" ) );
Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );

// the 'head' of the pipe assembly
Pipe assembly = new Pipe( "wordcount" );

// For each input Tuple
// parse out each word into a new Tuple with the field name "word"
// regular expressions are optional in Cascading
String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
Function function = new RegexGenerator( new Fields( "word" ), regex );
assembly = new Each( assembly, new Fields( "line" ), function );

// group the Tuple stream by the "word" value
assembly = new GroupBy( assembly, new Fields( "word" ) );

// For every Tuple group
// count the number of occurrences of "word" and store result in
// a field named "count"
Aggregator count = new Count( new Fields( "count" ) );
assembly = new Every( assembly, count );

// initialize app properties, tell Hadoop which jar file to use
Properties properties = AppProps.appProps()
  .setName( "word-count-application" )
  .setJarClass( Main.class )
  .buildProperties();

// plan a new Flow from the assembly using the source and sink Taps
// with the above properties
FlowConnector flowConnector = new HadoopFlowConnector( properties );
Flow flow = flowConnector.connect( "word-count", source, sink, assembly );

// execute the flow, block until complete
flow.complete();
```

Example 2.1 Word Counting

Several features of this example are worth highlighting.

First, notice that the pipe assembly is not coupled to the data (i.e., the `Tap` instances) until the last moment before execution. File paths or references are not embedded in the pipe assembly; instead, the pipe assembly is specified independent of data inputs and outputs. The only dependency is the data scheme, i.e., the field names. In Cascading,

every input or output file has field names associated with it, and every processing element of the pipe assembly either expects the specified fields or creates them. This allows developers to easily self-document their code, and allows the Cascading planner to "fail fast" if an expected dependency between elements isn't satisfied - for instance, if a needed field name is missing or incorrect. (If more information is desired on the planner, see MapReduce Job Planner.)

Also notice that pipe assemblies are assembled through constructor chaining. This may seem odd, but it is done for two reasons. First, it keeps the code more concise. Second, it prevents developers from creating "cycles" (i.e., recursive loops) in the resulting pipe assembly. Pipe assemblies are intended to be Directed Acyclic Graphs (DAG's), and in keeping with this, the Cascading planner is not designed to handle processes that feed themselves. (If desired, there are safer approaches to achieving this result.

Finally, notice that the very first `Pipe` instance has a name. That instance is the *head* of this particular pipe assembly. Pipe assemblies can have any number of heads, and any number of *tails*. Although the tail in this example does not have a name, in a more complex assembly it would. In general, heads and tails of pipe assemblies are assigned names to disambiguate them. One reason is that names are used to bind sources and sinks to pipes during planning. (The example above is an exception, because there is only one head and one tail - and consequently only one source and one sink - so the binding is unmistakable.) Another reason is that the naming of pipes contributes to self-documentation of pipe assemblies, especially where there are splits, joins, and merges in the assembly.

To sum up, the example word-counting application will:

- Read each line of text from a file and give it the field name "line"
- parse each "line" into words with the `RegexGenerator` object, which returns each word in the field named "word"
- sort and group all the tuples on the "word" field, using the `GroupBy` object
- count the number of elements in each group, using the `Count` object, and store this value in the "count" field
- and write out the "word" and "count" fields.

3. Data Processing

3.1 Terminology

The Cascading processing model is based on a metaphor of pipes (data streams) and filters (data operations). Thus the Cascading API allows the developer to assemble pipe assemblies that split, merge, group, or join streams of data while applying operations to each data record or groups of records.

In Cascading, we call a data record a *tuple*, a simple chain of pipes without forks or merges a *branch*, an interconnected set of pipe branches a *pipe assembly*, and a series of tuples passing through a pipe branch or assembly a *tuple stream*.

Pipe assemblies are specified independently of the data source they are to process. So before a pipe assembly can be executed, it must be bound to *taps*, i.e., data sources and sinks. The result of binding one or more pipe assemblies to taps is a *flow*, which is executed on a computer or cluster using the Hadoop framework.

Multiple flows can be grouped together and executed as a single process. In this context, if one flow depends on the output of another, it is not executed until all of its data dependencies are satisfied. Such a collection of flows is called a *cascade*.

3.2 Pipe Assemblies

Pipe assemblies define what work should be done against tuple streams, which are read from tap *sources* and written to tap *sinks*. The work performed on the data stream may include actions such as filtering, transforming, organizing, and calculating. Pipe assemblies may use multiple sources and multiple sinks, and may define splits, merges, and joins to manipulate the tuple streams.

Pipe Assembly Workflow

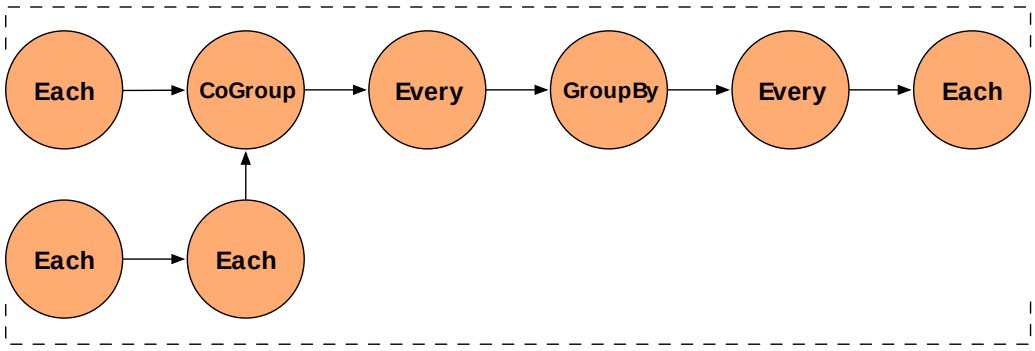
Pipe assemblies are created by chaining `cascading.pipe.Pipe` classes and subclasses together. Chaining is accomplished by passing the previous `Pipe` instances to the constructor of the next `Pipe` instance.

The following example demonstrates this type of chaining. It creates two pipes - a "left-hand side" (lhs) and a "right-hand side" (rhs) - and performs some processing on them both, using the Each pipe. Then it joins the two pipes into one, using the CoGroup pipe, and performs several operations on the joined pipe using Every and GroupBy. The specific operations performed are not important in the example; the point is to show the general flow of the data streams. The diagram after the example gives a visual representation of the workflow.

```
// the "left hand side" assembly head
Pipe lhs = new Pipe( "lhs" );

lhs = new Each( lhs, new SomeFunction() );
lhs = new Each( lhs, new SomeFilter() );

// the "right hand side" assembly head
Pipe rhs = new Pipe( "rhs" );

rhs = new Each( rhs, new SomeFunction() );

// joins the lhs and rhs
Pipe join = new CoGroup( lhs, rhs );

join = new Every( join, new SomeAggregator() );

join = new GroupBy( join );

join = new Every( join, new SomeAggregator() );

// the tail of the assembly
join = new Each( join, new SomeFunction() );
```

Example 3.1 Chaining Pipes

The following diagram is a visual representation of the example above.



Common Stream Patterns

As data moves through the pipe, streams may be separated or combined for various purposes. Here are the three basic patterns:

Split
: A split takes a single stream and sends it down multiple paths - that is, it feeds a single `Pipe` instance into two or more subsequent separate `Pipe` instances with unique branch names.

Merge
: A merge combines two or more streams that have identical fields into a single stream. This is done by passing two or more `Pipe` instances to a `Merge` or `GroupBy` pipe.

Join
: A join combines data from two or more streams that have different fields, based on common field values (analogous to a SQL join.) This is done by passing two or more `Pipe` instances to a `HashJoin` or `CoGroup` pipe. The code sequence and diagram above give an example.

Data Processing

In addition to directing the tuple streams - using splits, merges, and joins - pipe assemblies can examine, filter, organize, and transform the tuple data as the streams move through the pipe assemblies. To facilitate this, the values in the tuple are typically given field names, just as database columns are given names, so that they may be referenced or selected. The following terminology is used:

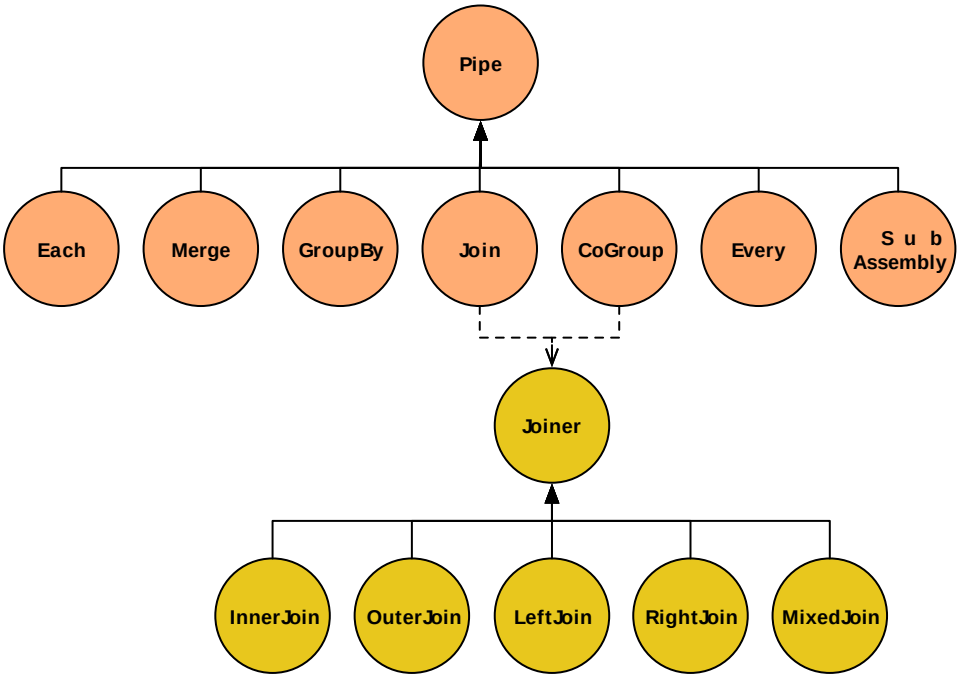
Operation
: Operations (`cascading.operation.Operation`) accept an input argument Tuple, and output zero or more result tuples. There are a few sub-types of operations defined below. Cascading has a number of generic Operations that can be used, or developers can create their own custom Operations.

Tuple
: In Cascading, data is processed as a stream of Tuples (`cascading.tuple.Tuple`), which are composed of fields, much like a database record or row. A Tuple is effectively an array of (field) values, where each value can be any `java.lang.Object` Java type (or `byte[]` array). For information on supporting non-primitive types, see Custom Types.

Fields
: Fields (`cascading.tuple.Fields`) are used either to declare the field names for fields in a Tuple, or reference field values in a Tuple. They can either be strings (such as "firstname" or "birthdate"), integers (for the field position, starting at `0` for the first position, or starting at `-1` for the last position), or one of the predefined *Fields sets* (such as `Fields.ALL`, which selects all values in the Tuple, like an asterisk in SQL). For more on Fields sets, see Field Algebra).

3.3 Pipes

The code for the sample pipe assembly above, Chaining Pipes, consists almost entirely of a series of `Pipe` constructors. This section describes the various `Pipe` classes in detail. The base class `cascading.pipe.Pipe` and its subclasses are shown in the diagram below.



Types of Pipes

The `Pipe` class is used to instantiate and name a pipe. Pipe names are used by the planner to bind taps to the pipe as sources or sinks. (A third option is to bind a tap to the pipe branch as a *trap*, discussed elsewhere as an advanced topic.)

The `SubAssembly` subclass is a special type of pipe. It is used to nest re-usable pipe assemblies within a `Pipe` class for inclusion in a larger pipe assembly. For more information on this, see the section on SubAssemblies.

The other six types of pipes are used to perform operations on the tuple streams as they pass through the pipe assemblies. This may involve operating on the individual tuples (e.g., transform or filter), on groups of related tuples (e.g., count or subtotal), or on entire streams (e.g., split, combine, group, or sort). These six pipe types are briefly introduced here, then explored in detail further below.

`Each`

These pipes perform operations based on the data contents of tuples - analyze, transform, or filter. The `Each` pipe operates on individual tuples in the stream, applying functions or filters such as conditionally replacing certain field values, removing tuples that have values outside a target range, etc.

You can also use `Each` to split or branch a stream, simply by routing the output of an `Each` into a different pipe or sink.

Note that with `Each`, as with other types of pipe, you can specify a list of fields to output, thereby removing unwanted fields from a stream.

`Merge`

Just as `Each` can be used to split one stream into two, `Merge` can be used to combine two or more streams into one, as long as they have the same fields.

A `Merge` accepts two or more streams that have identical fields, and emits a single stream of tuples (in arbitrary order) that contains all the tuples from all the specified input streams. Thus a Merge is just a mingling of all the tuples from the input streams, as if shuffling multiple card decks into one.

Use `Merge` when no grouping is required (i.e., no aggregator or buffer operations will be performed). `Merge` is much faster than `GroupBy` (see below) for merging.

To combine streams that have different fields, based on one or more common values, use `CoGroup` or `HashJoin`.

`GroupBy`

`GroupBy` groups the tuples of a stream based on common values in a specified field.

If passed multiple streams as inputs, it performs a merge before the grouping. As with `Merge`, a `GroupBy` requires that multiple input streams share the same field structure.

The purpose of grouping is typically to prepare a stream for processing by the `Every` pipe, which performs aggregator and buffer operations on the groups, such as counting, totalling, or averaging values within that group.

It should be clear that "grouping" here essentially means sorting all the tuples into groups based on the value of a particular field. However, within a given group, the tuples are in arbitrary order unless you specify a secondary sort key. For most purposes, a secondary sort is not required and only increases the execution time.

`Every`

The `Every` pipe operates on a tuple stream that has been grouped (by `GroupBy` or `CoGroup`) on the values of a particular field, such as timestamp or zipcode. It's used to apply aggregator or buffer operations such as counting, totaling, or averaging field values within each group. Thus the `Every` class is only for use on the output of `GroupBy` or `CoGroup`, and cannot be used with the output of `Each`, `Merge`, or `HashJoin`.

An `Every` instance may follow another `Every` instance, so `Aggregator` operations can be chained. This is not true for `Buffer` operations.

`CoGroup`

`CoGroup` performs a join on two or more streams, similar to a SQL join, and groups the single resulting output stream on the value of a specified field. As with SQL, the join can be inner, outer, left, or right. Self-joins are permitted, as well as mixed joins (for three or more streams) and custom joins. Null fields in the input streams become corresponding null fields in the output stream.

The resulting output stream contains fields from all the input streams. If the streams contain any field names in common, they must be renamed to avoid duplicate field names in the resulting tuples.

`HashJoin`

`HashJoin` performs a join on two or more streams, similar to a SQL join, and emits a single stream in arbitrary order. As with SQL, the join can be inner, outer, left, or right. Self-joins are permitted, as well as mixed joins (for three or more streams) and custom joins. Null fields in the input streams become corresponding null fields in the output stream.

For applications that do not require grouping, `HashJoin` provides faster execution than `CoGroup`, but only within certain prescribed cases. It is optimized for joining one or more small streams to no more than one

large stream. Developers should thoroughly understand the limitations of this class, as described below, before attempting to use it.

The following table summarizes the different types of pipes.

Table 3.1. Comparison of pipe types

Pipe type	Purpose	Input	Output
`Pipe`	instantiate a pipe; create or name a branch	name	a (named) pipe
`SubAssembly`	create nested subassemblies		
`Each`	apply a filter or function, or branch a stream	tuple stream (grouped or not)	a tuple stream, optionally filtered or transformed
`Merge`	merge two or more streams with identical fields	two or more tuple streams	a tuple stream, unsorted
`GroupBy`	sort/group on field values; optionally merge two or more streams with identical fields	one or more tuple streams with identical fields	a single tuple stream, grouped on key field(s) with optional secondary sort
`Every`	apply aggregator or buffer operation	grouped tuple stream	a tuple stream plus new fields with operation results
`CoGroup`	join 1 or more streams on matching field values	one or more tuple streams	a single tuple stream, joined on key field(s)
`HashJoin`	join 1 or more streams on matching field values	one or more tuple streams	a tuple stream in arbitrary order

The Each and Every Pipes

The `Each` and `Every` pipes perform operations on tuple data - for instance, perform a search-and-replace on tuple contents, filter out some of the tuples based on their contents, or count the number of tuples in a stream that share a common field value.

Here is the syntax for these pipes:

```
new Each( previousPipe, argumentSelector, operation, outputSelector )
```

```
new Every( previousPipe, argumentSelector, operation, outputSelector )
```

Both types take four arguments:

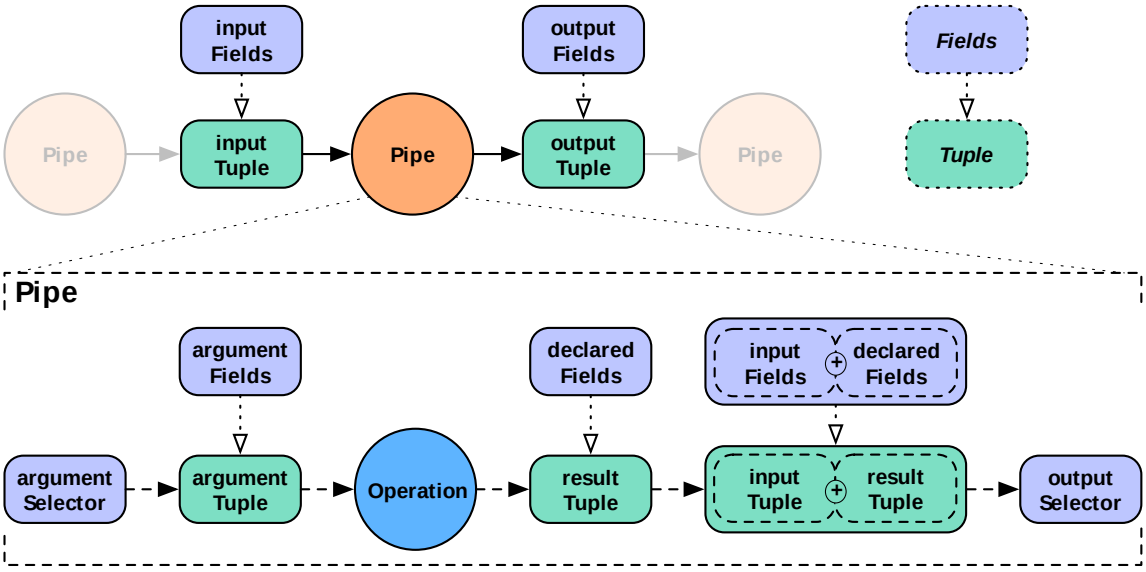
- a Pipe instance

- an argument selector
- an Operation instance
- an output selector on the constructor (selectors here are Fields instances)

The key difference between `Each` and `Every` is that the `Each` operates on individual tuples, and `Every` operates on groups of tuples emitted by `GroupBy` or `CoGroup`. This affects the kind of operations that these two pipes can perform, and the kind of output they produce as a result.

The `Each` pipe applies operations that are subclasses of `Functions` and `Filters` (described in the Javadoc). For example, using `Each` you can parse lines from a logfile into their constituent fields, filter out all lines except the HTTP GET requests, and replace the timestring fields with date fields.

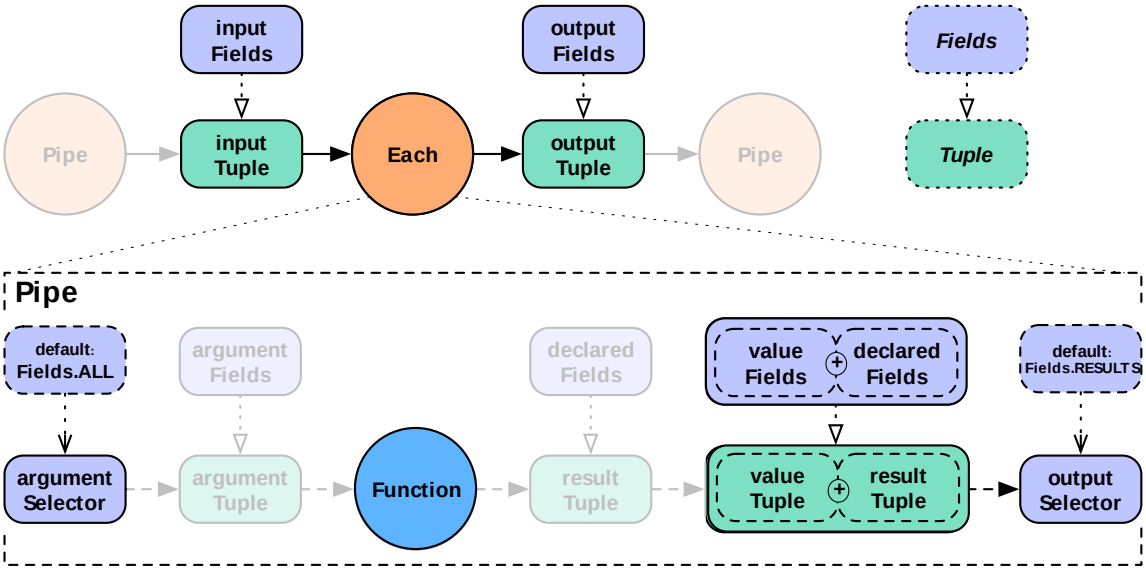
Similarly, since the `Every` pipe works on tuple groups (the output of a `GroupBy` or `CoGroup` pipe), it applies operations that are subclasses of `Aggregators` and `Buffers`. For example, you could use `GroupBy` to group the output of the above `Each` pipe by date, then use an `Every` pipe to count the GET requests per date. The pipe would then emit the operation results as the date and count for each group.



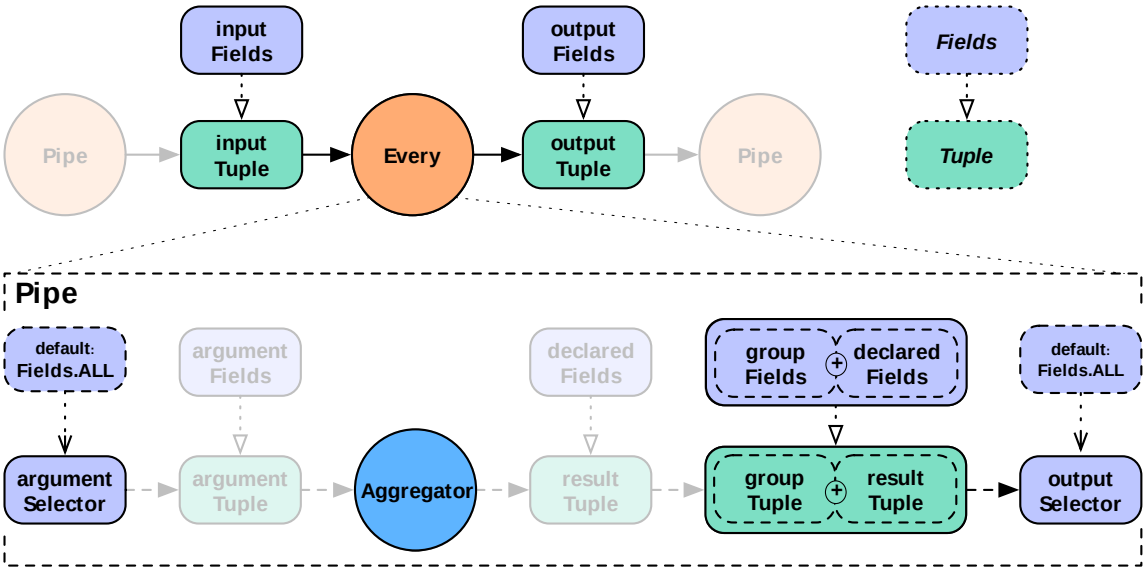
In the syntax shown at the start of this section, the *argument selector* specifies fields from the input tuple to use as input values. If the argument selector is not specified, the whole input tuple (`Fields.ALL`) is passed to the operation as a set of argument values.

Most `Operation` subclasses declare result fields (shown as "declared fields" in the diagram). The *output selector* specifies the fields of the output `Tuple` from the fields of the input `Tuple` and the operation result. This new output `Tuple` becomes the input `Tuple` to the next pipe in the pipe assembly. If the output selector is `Fields.ALL`, the output is the input `Tuple` plus the operation result, merged into a single `Tuple`.

Note that it's possible for a `Function` or `Aggregator` to return more than one output `Tuple` per input `Tuple`. In this case, the input tuple is duplicated as many times as necessary to create the necessary output tuples. This is similar to the reiteration of values that happens during a join. If a function is designed to always emit three result tuples for every input tuple, each of the three outgoing tuples will consist of the selected input tuple values plus one of the three sets of function result values.



If the result selector is not specified for an `Each` pipe performing a `Functions` operation, the operation results are returned by default (`Fields.RESULTS`), discarding the input tuple values in the tuple stream. (This is not true of `Filters` , which either discard the input tuple or return it intact, and thus do not use an output selector.)

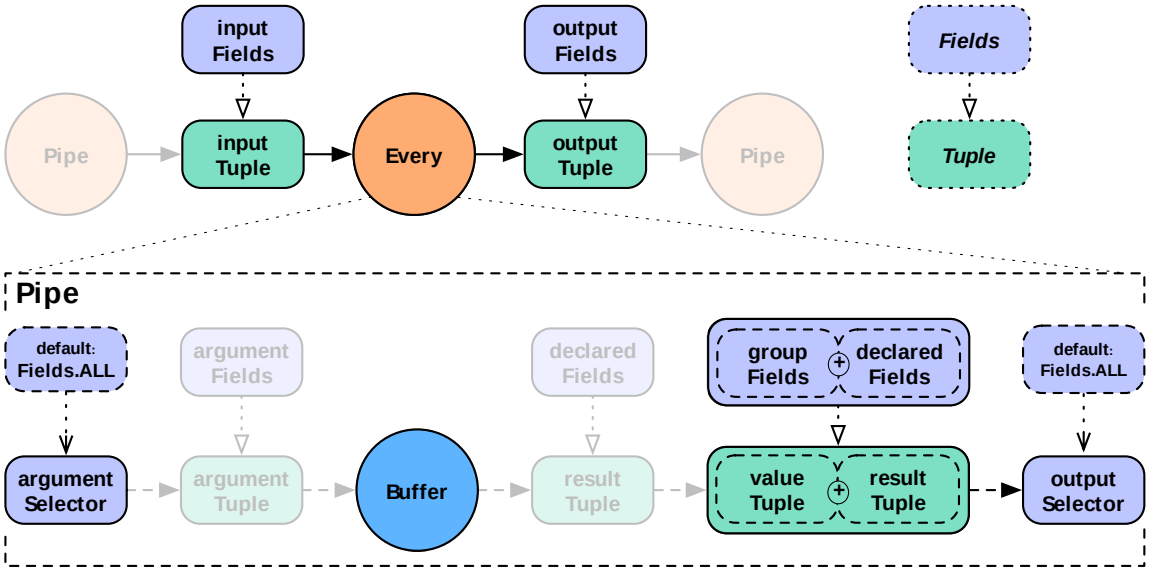


For the `Every` pipe, the Aggregator results are appended to the input Tuple (`Fields.ALL`) by default.

Note that the `Every` pipe associates `Aggregator` results with the current group `Tuple` (the unique keys currently being grouped on). For example, if you are grouping on the field "department" and counting the number of "names" grouped by that department, the resulting output Fields will be ["department","num_employees"].

If you are also adding up the salaries associated with each "name" in each "department", the output Fields will be ["department","num_employees","total_salaries"].

This is only true for chains of `Aggregator` Operations - you are not allowed to chain `Buffer` operations, as explained below.



When the `Every` pipe is used with a `Buffer` operation, instead of an `Aggregator`, the behavior is different. Instead of being associated with the current grouping tuple, the operation results are associated with the current values tuple. This is analogous to how an `Each` pipe works with a `Function`. This approach may seem slightly unintuitive, but provides much more flexibility. To put it another way, the results of the buffer operation are not appended to the current keys being grouped on. It is up to the buffer to emit them if they are relevant. It is also possible for a Buffer to emit more than one result Tuple per unique grouping. That is, a Buffer may or may not emulate an Aggregator, where an Aggregator is just a special optimized case of a Buffer.

For more information on how operations process fields, see Operations and Field-processing .

Merge

The `Merge` pipe is very simple. It accepts two or more streams that have the same fields, and emits a single stream containing all the tuples from all the input streams. Thus a merge is just a mingling of all the tuples from the input streams, as if shuffling multiple card decks into one. Note that the output of `Merge` is in arbitrary order.

```
Pipe merge = new Merge( lhs, rhs );
```

Example 3.2 Merging Two Tuple Streams

The example above simply combines all the tuples from two existing streams ("lhs" and "rhs") into a new tuple stream ("merge").

GroupBy

`GroupBy` groups the tuples of a stream based on common values in a specified field. If passed multiple streams as inputs, it performs a merge before the grouping. As with `Merge`, a `GroupBy` requires that multiple input streams share the same field structure.

The output of `GroupBy` is suitable for the `Every` pipe, which performs `Aggregator` and `Buffer` operations, such as counting, totalling, or averaging groups of tuples that have a common value (e.g., the same date). By default,

`GroupBy` performs no secondary sort, so within each group the tuples are in arbitrary order. For instance, when grouping on "lastname", the tuples `[doe, john]` and `[doe, jane]` end up in the same group, but in arbitrary sequence.

Secondary sorting

If multi-level sorting is desired, the names of the sort fields on must be specified to the `GroupBy` instance, as seen below. In this example, `value1` and `value2` will arrive in their natural sort order (assuming they are `java.lang.Comparable`).

```
Fields groupFields = new Fields( "group1", "group2" );
Fields sortFields = new Fields( "value1", "value2" );
Pipe groupBy = new GroupBy( assembly, groupFields, sortFields );
```

Example 3.3 Secondary Sorting

If we don't care about the order of `value2`, we can leave it out of the `sortFields` `Fields` constructor.

In the next example, we reverse the order of `value1` while keeping the natural order of `value2`.

```
Fields groupFields = new Fields( "group1", "group2" );
Fields sortFields = new Fields( "value1", "value2" );

sortFields.setComparator( "value1", Collections.reverseOrder() );

Pipe groupBy = new GroupBy( assembly, groupFields, sortFields );
```

Example 3.4 Reversing Secondary Sort Order

Whenever there is an implied sort during grouping or secondary sorting, a custom `java.util.Comparator` can optionally be supplied to the grouping `Fields` or secondary sort `Fields`. This allows the developer to use the `Fields.setComparator()` call to control the sort.

To sort or group on non-Java-comparable classes, consider creating a custom `Comparator`.

Below is a more practical example, where we group by the "day of the year", but want to reverse the order of the tuples within that grouping by "time of day".

```
Fields groupFields = new Fields( "year", "month", "day" );
Fields sortFields = new Fields( "hour", "minute", "second" );

sortFields.setComparators(
  Collections.reverseOrder(),   // hour
  Collections.reverseOrder(),   // minute
  Collections.reverseOrder() ); // second

Pipe groupBy = new GroupBy( assembly, groupFields, sortFields );
```

Example 3.5 Reverse Order by Time

CoGroup

The `CoGroup` pipe is similar to `GroupBy`, but instead of a merge, performs a join. That is, `CoGroup` accepts two or more input streams and groups them on one or more specified keys, and performs a join operation on equal key values, similar to a SQL join.

The output stream contains all the fields from all the input streams.

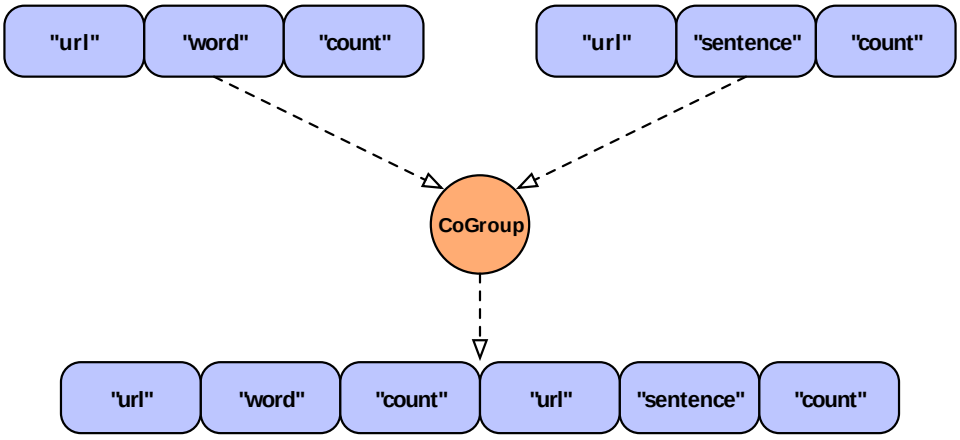
As with SQL, the join can be inner, outer, left, or right. Self-joins are permitted, as well as mixed joins (for three or more streams) and custom joins. Null fields in the input streams become corresponding null fields in the output stream.

Since the output is grouped, it is suitable for the `Every` pipe, which performs `Aggregator` and `Buffer` operations - such as counting, totalling, or averaging groups of tuples that have a common value (e.g., the same date).

The output stream is sorted by the natural order of the grouping fields. To control this order, at least the first `groupingFields` value given should be an instance of `Fields` containing `Comparator` instances for the appropriate fields. This allows fine-grained control of the sort grouping order.

Field names

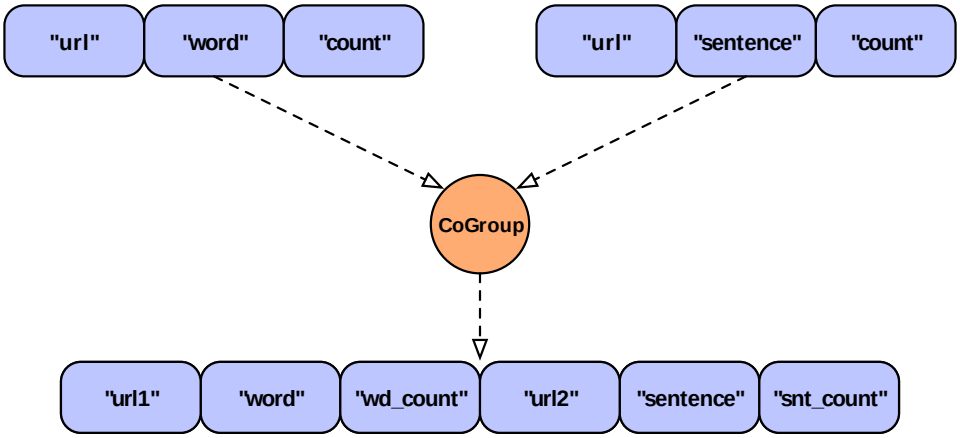
In a join operation, all the field names used in any of the input tuples must be unique; duplicate field names are not allowed. If the names overlap there is a collision, as shown in the following diagram.



In this figure, two streams are to be joined on the "url" field, resulting in a new Tuple that contains fields from the two input tuples. However, the resulting tuple would include two fields with the same name ("url"), which is unworkable. To handle the conflict, developers can use the *declaredFields* argument (described in the Javadoc) to declare unique field names for the output tuple, as in the following example.

```
Fields common = new Fields( "url" );
Fields declared = new Fields(
  "url1", "word", "wd_count", "url2", "sentence", "snt_count"
);
Pipe join =
  new CoGroup( lhs, common, rhs, common, declared, new InnerJoin() );
```

Example 3.6 Joining Two Tuple Streams with Duplicate Field Names



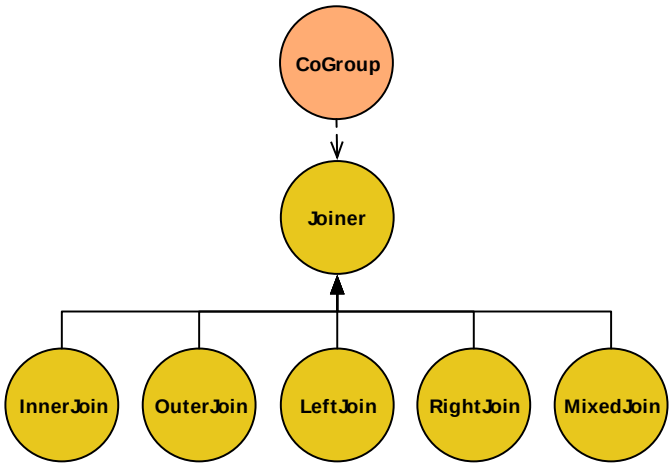
This revised figure demonstrates the use of declared field names to prevent a planning failure.

It might seem preferable for Cascading to automatically recognize the duplication and simply merge the identically-named fields, saving effort for the developer. However, consider the case of an outer type join in which one field (or set of fields used for the join) for a given join side happens to be `null`. Discarding one of the duplicate fields would lose this information.

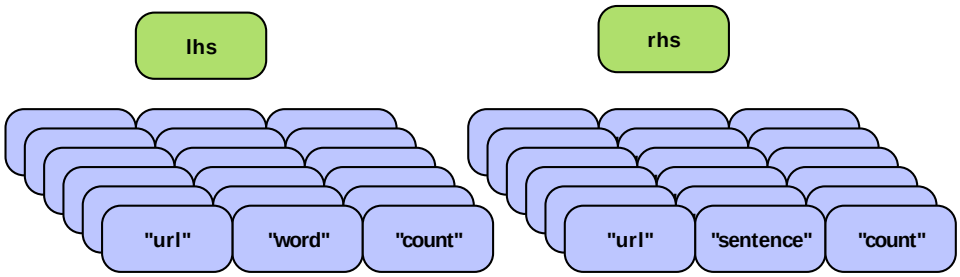
Further, the internal implementation relies on field position, not field names, when reading tuples; the field names are a device for the developer. This approach allows the behavior of the `CoGroup` to be deterministic and predictable.

The Joiner class

In the example above, we explicitly specified a Joiner class (InnerJoin) to perform a join on our data. There are five Joiner subclasses, as shown in this diagram.



In `CoGroup`, the join is performed after all the input streams are first co-grouped by their common keys. Cascading must create a "bag" of data for every grouping in the input streams, consisting of all the `Tuple` instances associated with that grouping.



It's already been mentioned that joins in Cascading are analogous to joins in SQL. The most commonly-used type of join is the inner join, the default in `CoGroup`. An inner join tries to match *each* Tuple on the "lhs" with *every* Tuple on the "rhs", based on matching field values. With an inner join, if either side has no tuples for a given value, no tuples are joined. An outer join, conversely, allows for either side to be empty and simply substitutes a `Tuple` containing `null` values for the non-existent tuple.

This sample data is used in the discussion below to explain and compare the different types of join:

```
LHS = [0,a] [1,b] [2,c]
RHS = [0,A] [2,C] [3,D]
```

In each join type below, the values are joined on the first tuple position (the join key), a numeric value. Note that, when Cascading joins tuples, the resulting `Tuple` contains all the incoming values from in incoming tuple streams, and does not discard the duplicate key fields. As mentioned above, on outer joins where there is no equivalent key in the alternate stream, `null` values are used.

For example using the data above, the result Tuple of an "inner" join with join key value of `2` would be `[2,c,2,C]`. The result Tuple of an "outer" join with join key value of `1` would be `[1,b,null,null]`.

InnerJoin

An inner join only returns a joined `Tuple` if neither bag for the join key is empty.

```
[0,a,0,A] [2,c,2,C]
```

OuterJoin
: An outer join performs a join if one bag (left or right) for the join key is empty, or if neither bag is empty.

```
[0,a,0,A] [1,b,null,null] [2,c,2,C] [null,null,3,D]
```

LeftJoin
: A left join can also be stated as a left inner and right outer join, where it is acceptable for the right bag to be empty (but not the left).

```
[0,a,0,A] [1,b,null,null] [2,c,2,C]
```

RightJoin
: A right join can also be stated as a left outer and right inner join, where it is acceptable for the left bag to be empty (but not the right).

```
[0,a,0,A] [2,c,2,C] [null,null,3,D]
```

MixedJoin
: A mixed join is where 3 or more tuple streams are joined, using a small Boolean array to specify each of the join types to use. For more information, see the `cascading.pipe.cogroup.MixedJoin` class in the Javadoc.

Custom
: Developers can subclass the `cascading.pipe.cogroup.Joiner` class to create custom join operations.

Scaling

`CoGroup` attempts to store the entire current unique keys tuple "bag" from the right-hand stream in memory for rapid joining to the left-hand stream. If the bag is very large, it may exceed a configurable threshold and be spilled to disk, reducing performance and potentially causing a memory error (if the threshold value is too large). Thus it's usually best to put the stream with the largest groupings on the left-hand side and, if necessary, adjust the spill threshold as described in the Javadoc.

HashJoin

`HashJoin` performs a join (similar to a SQL join) on two or more streams, and emits a stream of tuples that contain fields from all of the input streams. With a join, the tuples in the different input streams do not typically contain the same set of fields.

As with `CoGroup`, the field names must all be unique, including the names of the key fields, to avoid duplicate field names in the emitted `Tuple`. If necessary, use the *`declaredFields`* argument to specify unique field names for the output.

An inner join is performed by default, but you can choose inner, outer, left, right, or mixed (three or more streams). Self-joins are permitted. Developers can also create custom Joiners if desired. For more information on types of joins, refer to the section called “The Joiner class” or the Javadoc.

```
Fields lhsFields = new Fields( "fieldA", "fieldB" );
Fields rhsFields = new Fields( "fieldC", "fieldD" );
Pipe join =
  new HashJoin( lhs, lhsFields, rhs, rhsFields, new InnerJoin() );
```

Example 3.7 Joining Two Tuple Streams

The example above performs an inner join on two streams ("lhs" and "rhs"), based on common values in two fields. The field names that are specified in `lhsFields` and `rhsFields` are among the field names previously declared for the two input streams.

Scaling

For joins that do not require grouping, `HashJoin` provides faster execution than `CoGroup`, but it operates within stricter limitations. It is optimized for joining one or more small streams to no more than one large stream.

Unlike `CoGroup`, `HashJoin` attempts to keep the entire right-hand stream in memory for rapid comparison (not just the current grouping, as no grouping is performed for a `HashJoin`). Thus a very large tuple stream in the right-hand stream may exceed a configurable spill-to-disk threshold, reducing performance and potentially causing a memory error. For this reason, it's advisable to use the smaller stream on the right-hand side. Additionally, it may be helpful to adjust the spill threshold as described in the Javadoc.

Due to the potential difficulties of using `HashJoin` (as compared to the slower but much more reliable `CoGroup`), developers should thoroughly understand this class before attempting to use it.

Frequently the `HashJoin` is fed a filtered down stream of Tuples from what was originally a very large file. To prevent the large file from being replicated throughout a cluster, when running in Hadoop mode, use a `Checkpoint` pipe at the point where the data has been filtered down to its smallest before it is streamed into a `HashJoin`. This will force the Tuple stream to be persisted to disk and new `FlowStep` (MapReduce job) to be created to read the smaller data size more efficiently.

Setting Custom Pipe Properties

By default, the properties passed to a FlowConnector subclass become the defaults for every Flow instance created by that FlowConnector. In the past, if some of the Flow instances needed different properties, it was necessary to create additional FlowConnectors to set those properties. However, it is now possible to set properties at the Pipe scope and at the process FlowStep scope.

Setting properties at the Pipe scope lets you set a property that is only visible to a given Pipe instance (and its child Operation). This allows Operations such as custom Functions to be dynamically configured.

More importantly, setting properties at the process FlowStep scope allows you to set properties on a Pipe that are inherited by the underlying process during runtime. When running on the Apache Hadoop platform (i.e., when using the HadoopFlowConnector), a FlowStep is the current MapReduce job. Thus a Hadoop-specific property can be set on a Pipe, such as a CoGroup. During runtime, the FlowStep (MapReduce job) that the CoGroup executes in is configured with the given property - for example, a spill threshold, or the number of reducer tasks for Hadoop to deploy.

The following code samples demonstrates the basic form for both the Pipe scope and the process FlowStep scope.

```
Pipe join =
  new HashJoin( lhs, common, rhs, common, declared, new InnerJoin() );

SpillableProps props = SpillableProps.spillableProps()
  .setCompressSpill( true )
  .setMapSpillThreshold( 50 * 1000 );

props.setProperties( join.getConfigDef(), ConfigDef.Mode.REPLACE );
```

Example 3.8 Pipe Scope

```
Pipe join =
  new HashJoin( lhs, common, rhs, common, declared, new InnerJoin() );

SpillableProps props = SpillableProps.spillableProps()
  .setCompressSpill( true )
  .setMapSpillThreshold( 50 * 1000 );

props.setProperties( join.getStepConfigDef(), ConfigDef.Mode.DEFAULT );
```

Example 3.9 Step Scope

As of Cascading 2.2, SubAssemblies can now be configured via the ConfigDef method.

3.4 Platforms

Cascading supports pluggable planners that allow it to execute on differing platforms. Planners are invoked by an associated `FlowConnector` subclass. Currently, only two planners are provided, as described below:

LocalFlowConnector
: The `cascading.flow.local.LocalFlowConnector` provides a "local" mode planner for running Cascading completely in memory on the current computer. This allows for fast execution of Flows against local files or any other compatible custom `Tap` and `Scheme` classes.

  The local mode planner and platform were not designed to scale beyond available memory, CPU, or disk on the current machine. Thus any memory-intensive processes that use `GroupBy`, `CoGroup`, or `HashJoin` are likely to fail against moderately large files.

  Local mode is useful for development, testing, and interactive data exploration against sample sets.

HadoopFlowConnector
: The `cascading.flow.hadoop.HadoopFlowConnector` provides a planner for running Cascading on an Apache Hadoop 1 cluster. This allows Cascading to execute against extremely large data sets over a cluster of computing nodes.

Hadoop2MR1FlowConnector
: The `cascading.flow.hadoop2.Hadoop2MR1FlowConnector` provides a planner for running Cascading on an Apache Hadoop 2 cluster. This class is roughly equivalent to the above

`HadoopFlowConnector` except it uses Hadoop 2 specific properties and is compiled against Hadoop 2 API binaries.

Cascading's support for pluggable planners allows a pipe assembly to be executed on an arbitrary platform, using platform-specific Tap and Scheme classes that hide the platform-related I/O details from the developer. For example, Hadoop uses `org.apache.hadoop.mapred.InputFormat` to read data, but local mode is happy with a `java.io.FileInputStream`. This detail is hidden from developers unless they are creating custom Tap and Scheme classes.

3.5 Source and Sink Taps

All input data comes in from, and all output data goes out to, some instance of `cascading.tap.Tap`. A tap represents a data resource - such as a file on the local file system, on a Hadoop distributed file system, or on Amazon S3. A tap can be read from, which makes it a *source*, or written to, which makes it a *sink*. Or, more commonly, taps act as both sinks and sources when shared between flows.

The platform on which your application is running (Cascading local or Hadoop) determines which specific classes you can use. Details are provided in the sections below.

Schemes

If the Tap is about where the data is and how to access it, the Scheme is about what the data is and how to read it. Every Tap must have a Scheme that describes the data. Cascading provides four Scheme classes:

TextLine
: `TextLine` reads and writes raw text files and returns tuples which, by default, contain two fields specific to the platform used. The first field is either the byte offset or line number, and the second field is the actual line of text. When written to, all Tuple values are converted to Strings delimited with the TAB character (\t). A TextLine scheme is provided for both the local and Hadoop modes.

  By default TextLine uses the UTF-8 character set. This can be overridden on the appropriate TextLine constructor.

TextDelimited
: `TextDelimited` reads and writes character-delimited files in standard formats such as CSV (comma-separated variables), TSV (tab-separated variables), and so on. When written to, all Tuple values are converted to Strings and joined with the specified character delimiter. This Scheme can optionally handle quoted values with custom quote characters. Further, TextDelimited can coerce each value to a primitive type when reading a text file. A TextDelimited scheme is provided for both the local and Hadoop modes.

  By default TextDelimited uses the UTF-8 character set. This can be overridden on appropriate the TextDelimited constructor.

SequenceFile
: `SequenceFile` is based on the Hadoop Sequence file, which is a binary format. When written to or read from, all Tuple values are saved in their native binary form. This is the most efficient file format - but be aware that the resulting files are binary and can only be read by Hadoop applications running on the Hadoop platform.

WritableSequenceFile
: Like the `SequenceFile` Scheme, `WritableSequenceFile` is based on the Hadoop Sequence file, but it was designed to read and write key and/or value Hadoop `Writable` objects directly. This is very useful if you

have sequence files created by other applications. During writing (sinking), specified key and/or value fields are serialized directly into the sequence file. During reading (sourcing), the key and/or value objects are deserialized and wrapped in a Cascading Tuple object and passed to the downstream pipe assembly. This class is only available when running on the Hadoop platform.

There's a key difference between the `TextLine` and `SequenceFile` schemes. With the `SequenceFile` scheme, data is stored as binary tuples, which can be read without having to be parsed. But with the `TextLine` option, Cascading must parse each line into a `Tuple` before processing it, causing a performance hit.

Platform-specific implementation details

Depending on which platform you use (Cascading local or Hadoop), the classes you use to specify schemes will vary. Platform-specific details for each standard scheme are shown below.

Table 3.2. Platform-specific tap scheme classes

Description	**Cascading local platform**	**Hadoop platform**
Package Name	`cascading.scheme.local`	`cascading.scheme.hadoop`
Read lines of text	`TextLine`	`TextLine`
Read delimited text (CSV, TSV, etc)	`TextDelimited`	`TextDelimited`
Cascading proprietary efficient binary		`SequenceFile`
External Hadoop application binary (custom `Writable` type)		`WritableSequenceFile`

Sequence File Compression

For best performance when running on the Hadoop platform, enable Sequence File Compression in the Hadoop property settings - either block or record-based compression. Refer to the Hadoop documentation for the available properties and compression types.

Taps

The following sample code creates a new Hadoop FileSystem Tap that can read and write raw text files. Since only one field name is provided, the "offset" field is discarded, resulting in an input tuple stream with only "line" values.

```
Tap tap = new Hfs( new TextLine( new Fields( "line" ) ), path );
```

Example 3.10 Creating a new tap

Here are the most commonly-used tap types:

FileTap
: The `cascading.tap.local.FileTap` tap is used with the Cascading local platform to access files on the local file system.

Hfs
: The `cascading.tap.hadoop.Hfs` tap uses the current Hadoop default file system, when running on the Hadoop platform.

  If Hadoop is configured for "Hadoop local mode" (not to be confused with Cascading local mode), its default file system is the local file system. If configured for distributed mode, its default file system is typically the Hadoop distributed file system.

  Note that Hadoop can be forced to use an external file system by specifying a prefix to the URL passed into a new Hfs tap. For instance, using "s3://somebucket/path" tells Hadoop to use the S3 `FileSystem` implementation to access files in an Amazon S3 bucket. More information on this can be found in the Javadoc.

Also provided are six utility taps:

MultiSourceTap
: The `cascading.tap.MultiSourceTap` is used to tie multiple tap instances into a single tap for use as an input source. The only restriction is that all the tap instances passed to a new MultiSourceTap share the same Scheme classes (not necessarily the same Scheme instance).

MultiSinkTap
: The `cascading.tap.MultiSinkTap` is used to tie multiple tap instances into a single tap for use as output sinks. At runtime, for every Tuple output by the pipe assembly, each child tap to the MultiSinkTap will sink the Tuple.

PartitionTap
: The `cascading.tap.hadoop.PartitionTap` and `cascading.tap.local.PartitionTap` are used to sink tuples into directory paths based on the values in the Tuple. More can be read below in Partition Taps. Note the `TemplateTap` has been deprecated in favor of the `PartitionTap`.

GlobHfs
: The `cascading.tap.hadoop.GlobHfs` tap accepts Hadoop style "file globbing" expression patterns. This allows for multiple paths to be used as a single source, where all paths match the given pattern. This tap is only available when running on the Hadoop platform.

DecoratorTap
: The `cascading.tap.DecoratorTap` is a utility helper for wrapping an existing Tap with new functionality, via sub-class, and/or adding 'meta-data' to a Tap instance via the generic `MetaInfo` instance field. Further, on the Hadoop platform, planner created intermediate and `Checkpoint` Taps can be wrapped by a `DecoratorTap` implementation by the Cascading Planner. See `cascading.flow.FlowConnectorProps` for details.

DistCacheTap
: The `cascading.tap.hadoop.DistCacheTap` is a sub-class of the `cascading.tap.DecoratorTap` that can wrap an cascading.tap.hadoop.Hfs instance. It allows for writing to HDFS, but reading from the Hadoop Distributed Cache under the write circumstances, specifically if the Tap is being read into the small side of a `cascading.pipe.HashJoin`.

Platform-specific implementation details

Depending on which platform you use (Cascading local or Hadoop), the classes you use to specify file systems will vary. Platform-specific details for each standard tap type are shown below.

Table 3.3. Platform-specific details for setting file system

Description	Either platform	Cascading local platform	Hadoop platform
Package Name	`cascading.tap`	`cascading.tap.local`	`cascading.tap.hadoop`
File access		`FileTap`	`Hfs`
Multiple Taps as single source	`MultiSourceTap`		
Multiple Taps as single sink	`MultiSinkTap`		
Bin/Partition data into multiple files		`PartitionTap`	`PartitionTap`
Pattern match multiple files/dirs			`GlobHfs`
Wrapping a Tap with MetaData / Decorating intra-Flow Taps	`DecoratorTap`		
Reading from the Hadoop Distributed Cache			`DistCacheTap`

3.6 Sink modes

```
Tap tap =
  new Hfs( new TextLine( new Fields( "line" ) ), path, SinkMode.REPLACE );
```

Example 3.11 Overwriting An Existing Resource

All applications created with Cascading read data from one or more sources, process it, then write data to one or more sinks. This is done via the various `Tap` classes, where each class abstracts different types of back-end systems that store data as files, tables, blobs, and so on. But in order to sink data, some systems require that the resource (e.g., a file) not exist before processing thus must be removed (deleted) before the processing can begin. Other systems may allow for appending or updating of a resource (typical with database tables).

When creating a new `Tap` instance, a `SinkMode` may be provided so that the Tap will know how to handle any existing resources. Note that not all Taps support all `SinkMode` values - for example, Hadoop does not support appends (updates) from a MapReduce job.

The available SinkModes are:

`SinkMode.KEEP`
: This is the default behavior. If the resource exists, attempting to write over it will fail.

`SinkMode.REPLACE`
: This allows Cascading to delete the file immediately after the Flow is started.

`SinkMode.UPDATE`
: Allows for new tap types that can update or append - for example, to update or add records in a database. Each tap may implement this functionality in its own way. Cascading recognizes this update mode, and if a resource exists, will not fail or attempt to delete it.

Note that Cascading itself only uses these labels internally to know when to automatically call `deleteResource()` on the `Tap` or to leave the Tap alone. It is up the the `Tap` implementation to actually perform a write or update when processing starts. Thus, when `start()` or `complete()` is called on a `Flow`, any sink `Tap` labeled `SinkMode.REPLACE` will have its `deleteResource()` method called.

Conversely, if a `Flow` is in a `Cascade` and the `Tap` is set to `SinkMode.KEEP` or `SinkMode.REPLACE`, `deleteResource()` will be called if and only if the sink is stale (i.e., older than the source). This allows a `Cascade` to behave like a "make" or "ant" build file, only running Flows that should be run. For more information, see Skipping Flows.

It's also important to understand how Hadoop deals with directories. By default, Hadoop cannot source data from directories with nested sub-directories, and it cannot write to directories that already exist. However, the good news is that you can simply point the `Hfs` tap to a directory of data files, and they are all used as input - there's no need to enumerate each individual file into a `MultiSourceTap`. If there are nested directories, use `GlobHfs`.

3.7 Fields Sets

Cascading applications can perform complex manipulation or "field algebra" on the fields stored in tuples, using *Fields sets*, a feature of the `Fields` class that provides a sort of wildcard tool for referencing sets of field values.

These predefined Fields sets are constant values on the `Fields` class. They can be used in many places where the `Fields` class is expected. They are:

Fields.ALL
: The `cascading.tuple.Fields.ALL` constant is a wildcard that represents all the current available fields.

```
// incoming -> first, last, age

String expression = "first + \" \" + last";
Fields fields = new Fields( "full" );
ExpressionFunction full =
  new ExpressionFunction( fields, expression, String.class );

assembly =
  new Each( assembly, new Fields( "first", "last" ), full, Fields.ALL );

// outgoing -> first, last, age, full
```

Fields.RESULTS

The `cascading.tuple.Fields.RESULTS` constant is used to represent the field names of the current operations return values. This Fields set may only be used as an output selector on a pipe, causing the pipe to output a tuple containing the operation results.

```
// incoming -> first, last, age

String expression = "first + \" \" + last";
Fields fields = new Fields( "full" );
ExpressionFunction full =
  new ExpressionFunction( fields, expression, String.class );

Fields firstLast = new Fields( "first", "last" );
assembly =
  new Each( assembly, firstLast, full, Fields.RESULTS );

// outgoing -> full
```

Fields.REPLACE

The `cascading.tuple.Fields.REPLACE` constant is used as an output selector to inline-replace values in the incoming tuple with the results of an operation. This convenient Fields set allows operations to overwrite the value stored in the specified field. The current operation must either specify the identical argument selector field names used by the pipe, or use the ARGS Fields set.

```
// incoming -> first, last, age

// coerce to int
Identity function = new Identity( Fields.ARGS, Integer.class );

Fields age = new Fields( "age" );
assembly = new Each( assembly, age, function, Fields.REPLACE );

// outgoing -> first, last, age
```

Fields.SWAP

The `cascading.tuple.Fields.SWAP` constant is used as an output selector to swap the operation arguments with its results. Neither the argument and result field names, nor the size, need to be the same. This is useful for when the operation arguments are no longer necessary and the result Fields and values should be appended to the remainder of the input field names and Tuple.

```
// incoming -> first, last, age

String expression = "first + \" \" + last";
Fields fields = new Fields( "full" );
ExpressionFunction full =
  new ExpressionFunction( fields, expression, String.class );
```

```
Fields firstLast = new Fields( "first", "last" );
assembly = new Each( assembly, firstLast, full, Fields.SWAP );

// outgoing -> age, full
```

Fields.ARGS

The `cascading.tuple.Fields.ARGS` constant is used to let a given operation inherit the field names of its argument Tuple. This Fields set is a convenience and is typically used when the Pipe output selector is `RESULTS` or `REPLACE`. It is specifically used by the Identity Function when coercing values from Strings to primitive types.

```
// incoming -> first, last, age

// coerce to int
Identity function = new Identity( Fields.ARGS, Integer.class );

Fields age = new Fields( "age" );
assembly = new Each( assembly, age, function, Fields.REPLACE );

// outgoing -> first, last, age
```

Fields.GROUP

The `cascading.tuple.Fields.GROUP` constant represents all the fields used as grouping key in the most recent grouping. If no previous grouping exists in the pipe assembly, `GROUP` represents all the current field names.

```
// incoming -> first, last, age

assembly = new GroupBy( assembly, new Fields( "first", "last" ) );

FieldJoiner full = new FieldJoiner( new Fields( "full" ), " " );

assembly = new Each( assembly, Fields.GROUP, full, Fields.ALL );

// outgoing -> first, last, age, full
```

Fields.VALUES

The `cascading.tuple.Fields.VALUES` constant represents all the fields not used as grouping fields in a previous Group. That is, if you have fields "a", "b", and "c", and group on "a", `Fields.VALUES` will resolve to "b" and "c".

```
// incoming -> first, last, age

assembly = new GroupBy( assembly, new Fields( "age" ) );

FieldJoiner full = new FieldJoiner( new Fields( "full" ), " " );
```

```
assembly = new Each( assembly, Fields.VALUES, full, Fields.ALL );

// outgoing -> first, last, age, full
```

Fields.UNKNOWN

The `cascading.tuple.Fields.UNKNOWN` constant is used when Fields must be declared, but it's not known how many fields or what their names are. This allows for processing tuples of arbitrary length from an input source or some operation. Use this Fields set with caution.

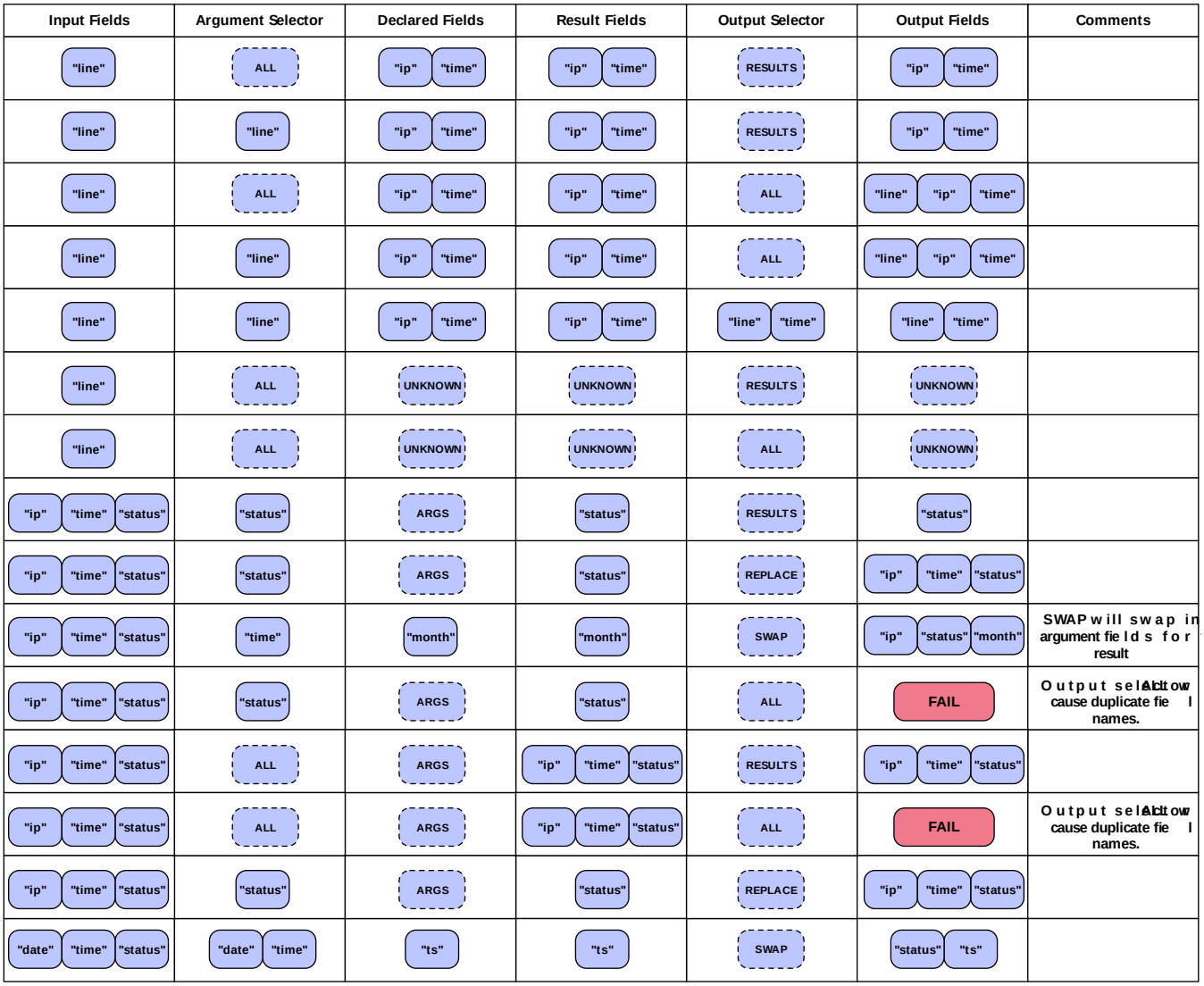
```
// incoming -> line

RegexSplitter function = new RegexSplitter( Fields.UNKNOWN, "\t" );

Fields fields = new Fields( "line" );
assembly =
  new Each( assembly, fields, function, Fields.RESULTS );

// outgoing -> unknown
```

Fields.NONE

The `cascading.tuple.Fields.NONE` constant is used to specify no fields. Typically used as an argument selector for Operations that do not process any Tuples, like `cascading.operation.Insert`.

```
// incoming -> first, last, age

Insert constant = new Insert( new Fields( "zip" ), "77373" );

assembly = new Each( assembly, Fields.NONE, constant, Fields.ALL );

// outgoing -> first, last, age, zip
```

The chart below shows common ways to merge input and result fields for the desired output fields. A few minutes with this chart may help clarify the discussion of fields, tuples, and pipes. Also see Each and Every Pipes for details on the different columns and their relationships to the `Each` and `Every` pipes and Functions, Aggregators, and Buffers.



3.8 Flows

When pipe assemblies are bound to source and sink taps, a `Flow` is created. Flows are executable in the sense that, once they are created, they can be started and will execute on the specified platform. If the Hadoop platform is specified, the Flow will execute on a Hadoop cluster.

A Flow is essentially a data processing pipeline that reads data from sources, processes the data as defined by the pipe assembly, and writes data to the sinks. Input source data does not need to exist at the time the Flow is created, but it must exist by the time the Flow is executed (unless it is executed as part of a Cascade - see Cascades for more on this).

The most common pattern is to create a Flow from an existing pipe assembly. But there are cases where a MapReduce job (if running on Hadoop) has already been created, and it makes sense to encapsulate it in a Flow class so that it may participate in a `Cascade` and be scheduled with other `Flow` instances. Alternatively, via the Riffle [http://github.com/cwensel/riffle] annotations, third-party applications can participate in a `Cascade`, and complex algorithms that result in iterative Flow executions can be encapsulated as a single Flow. All patterns are covered here.

Creating Flows from Pipe Assemblies

```
HadoopFlowConnector flowConnector = new HadoopFlowConnector();

Flow flow =
  flowConnector.connect( "flow-name", source, sink, pipe );
```

Example 3.12 Creating a new Flow

To create a Flow, it must be planned though one of the FlowConnector subclass objects. In Cascading, each platform (i.e., local and Hadoop) has its own connectors. The `connect()` method is used to create new Flow instances based on a set of sink taps, source taps, and a pipe assembly. Above is a trivial example that uses the Hadoop mode connector.

```
// the "left hand side" assembly head
Pipe lhs = new Pipe( "lhs" );

lhs = new Each( lhs, new SomeFunction() );
lhs = new Each( lhs, new SomeFilter() );

// the "right hand side" assembly head
Pipe rhs = new Pipe( "rhs" );

rhs = new Each( rhs, new SomeFunction() );

// joins the lhs and rhs
Pipe join = new CoGroup( lhs, rhs );

join = new Every( join, new SomeAggregator() );

Pipe groupBy = new GroupBy( join );

groupBy = new Every( groupBy, new SomeAggregator() );

// the tail of the assembly
groupBy = new Each( groupBy, new SomeFunction() );

Tap lhsSource = new Hfs( new TextLine(), "lhs.txt" );
Tap rhsSource = new Hfs( new TextLine(), "rhs.txt" );

Tap sink = new Hfs( new TextLine(), "output" );

FlowDef flowDef = new FlowDef()
  .setName( "flow-name" )
  .addSource( rhs, rhsSource )
  .addSource( lhs, lhsSource )
  .addTailSink( groupBy, sink );

Flow flow = new HadoopFlowConnector().connect( flowDef );
```

Example 3.13 Binding taps in a Flow

The example above expands on our previous pipe assembly example by creating multiple source and sink taps and planning a Flow. Note there are two branches in the pipe assembly - one named "lhs" and the other named "rhs". Internally Cascading uses those names to bind the source taps to the pipe assembly. New in 2.0, a FlowDef can be created to manage the names and taps that must be passed to a FlowConnector.

Configuring Flows

The FlowConnector constructor accepts the `java.util.Property` object so that default Cascading and any platform-specific properties can be passed down through the planner to the platform at runtime. In the case of Hadoop, any relevant Hadoop `*-default.xml` properties may be added. For instance, it's very common to add `mapred.map.tasks.speculative.execution`, `mapred.reduce.tasks.speculative.execution`, or `mapred.child.java.opts`.

One of the two properties that must always be set for production applications is the application Jar class or Jar path.

```
Properties properties = new Properties();

// pass in the class name of your application
// this will find the parent jar at runtime
properties = AppProps.appProps()
  .setName( "sample-app" )
  .setVersion( "1.2.3" )
  .addTags( "deploy:prod", "team:engineering" )
  .setJarClass( Main.class ) // find jar from class
  .buildProperties( properties ); // returns a copy

// ALTERNATIVELY ...

// pass in the path to the parent jar
properties = AppProps.appProps()
  .setName( "sample-app" )
  .setVersion( "1.2.3" )
  .addTags( "deploy:prod", "team:engineering" )
  .setJarPath( pathToJar ) // set jar path
  .buildProperties( properties ); // returns a copy


// pass properties to the connector
FlowConnector flowConnector = new HadoopFlowConnector( properties );
```

Example 3.14 Configuring the Application Jar

More information on packaging production applications can be found in Executing Processes.

Since the `FlowConnector` can be reused, any properties passed on the constructor will be handed to all the Flows it is used to create. If Flows need to be created with different default properties, a new FlowConnector will need to be instantiated with those properties, or properties will need to be set on a given `Pipe` or `Tap` instance directly - via the `getConfigDef()` or `getStepConfigDef()` methods.

Skipping Flows

When a `Flow` participates in a `Cascade`, the `Flow.isSkipFlow()` method is consulted before calling `Flow.start()` on the flow. The result is based on the Flow's *skip strategy*. By default, `isSkipFlow()` returns true if any of the sinks are stale - i.e., the sinks don't exist or the resources are older than the sources. However, the strategy can be changed via the `Flow.setFlowSkipStrategy()` and `Cascade.setFlowSkipStrategy()` method, which can be called before or after a particular `Flow` instance has been created.

Cascading provides a choice of two standard skip strategies:

FlowSkipIfSinkNotStale
: This strategy - `cascading.flow.FlowSkipIfSinkNotStale` - is the default. Sinks are treated as stale if they don't exist or the sink resources are older than the sources. If the SinkMode for the sink tap is REPLACE, then the tap is treated as stale.

FlowSkipIfSinkExists
: The `cascading.flow.FlowSkipIfSinkExists` strategy skips the Flow if the sink tap exists, regardless of age. If the `SinkMode` for the sink tap is `REPLACE`, then the tap is treated as stale.

Additionally, you can implement custom skip strategies by using the interface `cascading.flow.FlowSkipStrategy`.

Note that `Flow.start()` does not consult the `isSkipFlow()` method, and consequently always tries to start the Flow if called. It is up to the user code to call `isSkipFlow()` to determine whether the current strategy indicates that the Flow should be skipped.

Creating Flows from a JobConf

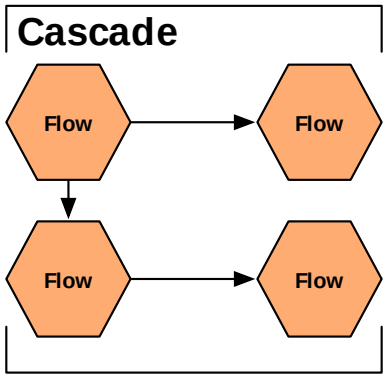
If a MapReduce job already exists and needs to be managed by a Cascade, then the `cascading.flow.hadoop.MapReduceFlow` class should be used. To do this, after creating a Hadoop `JobConf` instance simply pass it into the `MapReduceFlow` constructor. The resulting `Flow` instance can be used like any other Flow.

Creating Custom Flows

Any custom Class can be treated as a Flow if given the correct Riffle [http://github.com/cwensel/riffle] annotations. Riffle is a set of Java annotations that identify specific methods on a class as providing specific life-cycle and dependency functionality. For more information, see the Riffle documentation and examples. To use with Cascading, a Riffle-annotated instance must be passed to the `cascading.flow.hadoop.ProcessFlow` constructor method. The resulting `ProcessFlow` instance can be used like any other Flow instance.

Since many algorithms need to perform multiple passes over a given data set, a Riffle-annotated Class can be written that internally creates Cascading Flows and executes them until no more passes are needed. This is like nesting Flows or Cascades in a parent Flow, which in turn can participate in a Cascade.

3.9 Cascades



A Cascade allows multiple Flow instances to be executed as a single logical unit. If there are dependencies between the Flows, they are executed in the correct order. Further, Cascades act like Ant builds or Unix make files - that is, a Cascade only executes Flows that have stale sinks (i.e., output data that is older than the input data). For more on this, see Skipping Flows.

```
CascadeConnector connector = new CascadeConnector();
Cascade cascade = connector.connect( flowFirst, flowSecond, flowThird );
```

Example 3.15 Creating a new Cascade

When passing Flows to the CascadeConnector, order is not important. The CascadeConnector automatically identifies the dependencies between the given Flows and creates a scheduler that starts each Flow as its data sources become available. If two or more Flow instances have no interdependencies, they are submitted together so that they can execute in parallel.

For more information, see the section on Topological Scheduling.

If an instance of `cascading.flow.FlowSkipStrategy` is given to a `Cascade` instance (via the `Cascade.setFlowSkipStrategy()` method), it is consulted for every Flow instance managed by that Cascade, and all skip strategies on those Flow instances are ignored. For more information on skip strategies, see Skipping Flows.

4. Executing Processes on Hadoop

4.1 Introduction

This section covers some of the operational mechanics of running an application that uses Cascading with the Hadoop platform, including building the application jar file and configuring the operating mode.

To use the `HadoopFlowConnector` (i.e., to run in Hadoop mode), Cascading requires that Apache Hadoop be installed and correctly configured. Hadoop is an Open Source Apache project, freely available for download from the Hadoop website, http://hadoop.apache.org/core/.

4.2 Building

Cascading ships with several jars and dependencies in the download archive. Alternatively, Cascading is available over Maven and Ivy through the Conjars repository, along with a number of other Cascading-related projects. See http://conjars.org [http://conjars.org/] for more information.

The core Cascading artifacts include the following:

cascading-core-2.6.x.jar
: This jar contains the Cascading Core class files. It should be packaged with `lib/*.jar` when using Hadoop.

cascading-local-2.6.x.jar
: This jar contains the Cascading local mode class files. It is not needed when using Hadoop.

cascading-hadoop-2.6.x.jar
: This jar contains the Cascading Hadoop 1 specific dependencies. It should be packaged with `lib/*.jar` when using Hadoop.

cascading-hadoop2-mr1-2.6.x.jar
: This jar contains the Cascading Hadoop 2 specific dependencies. It should be packaged with `lib/*.jar` when using Hadoop.

cascading-xml-2.6.x.jar
: This jar contains Cascading XML module class files and is optional. It should be packaged with `lib/xml/*.jar` when using Hadoop.

Cascading works with either of the Hadoop processing modes - the default local standalone mode and the distributed cluster mode. As specified in the Hadoop documentation, running in cluster mode requires the creation of a Hadoop job jar that includes the Cascading jars, plus any needed third-party jars, in its `lib` directory. This is true regardless of whether they are Cascading Hadoop-mode applications or raw Hadoop MapReduce applications.

4.3 Configuring

During runtime, Hadoop must be told which application jar file should be pushed to the cluster. Typically, this is done via the Hadoop API `JobConf` object.

Cascading offers a shorthand for configuring this parameter, demonstrated here:

```
Properties properties = new Properties();

// pass in the class name of your application
// this will find the parent jar at runtime
properties = AppProps.appProps()
  .setName( "sample-app" )
  .setVersion( "1.2.3" )
  .addTags( "deploy:prod", "team:engineering" )
  .setJarClass( Main.class ) // find jar from class
  .buildProperties( properties ); // returns a copy

// ALTERNATIVELY ...

// pass in the path to the parent jar
properties = AppProps.appProps()
  .setName( "sample-app" )
  .setVersion( "1.2.3" )
  .addTags( "deploy:prod", "team:engineering" )
  .setJarPath( pathToJar ) // set jar path
  .buildProperties( properties ); // returns a copy


// pass properties to the connector
FlowConnector flowConnector = new HadoopFlowConnector( properties );
```

Above we see two ways to set the same property - via the `setJarClass()` method, and via the `setJarPath()` method. One is based on a Class name, and the other is based on a literal path.

The first method takes a Class object that owns the "main" function for this application. The assumption here is that `Main.class` is not located in a Java Jar that is stored in the `lib` folder of the application Jar. If it is, that Jar is pushed to the cluster, not the parent application jar.

The second method simply sets the path to the Java Jar as a property.

In your application, only one of these methods needs to be called, but one of them must be called to properly configure Hadoop.

```
JobConf jobConf = new JobConf();

// pass in the class name of your application
// this will find the parent jar at runtime
jobConf.setJarByClass( Main.class );

// ALTERNATIVELY ...

// pass in the path to the parent jar
jobConf.setJar( pathToJar );

// build the properties object using jobConf as defaults
Properties properties = AppProps.appProps()
  .setName( "sample-app" )
  .setVersion( "1.2.3" )
  .addTags( "deploy:prod", "team:engineering" )
  .buildProperties( jobConf );

// pass properties to the connector
FlowConnector flowConnector = new HadoopFlowConnector( properties );
```

Example 4.1 Configuring the Application Jar with a JobConf

Above we are starting with an existing Hadoop `JobConf` instance and building a Properties object with it as the default.

Note that `AppProps` is a helper fluent API for setting properties that define Flows or configure the underlying platform. There are quite a few "Props" based classes that expose fluent API calls, the ones most commonly used are below.

`cascading.property.AppProps`	Allows for setting application specific properties. Some properties are required by the underlying platform, like application Jar. Others are simple meta-data used by compatible management tools, like tags.
`cascading.flow.FlowConnectorProps`	Allows for setting `DebugLevel` or `AssertionLevel` for a given FlowConnector to target. Also allows for setting intermediate `DecoratorTap` sub-classes to be used if any.
`cascading.flow.FlowProps`	Allows for setting any Flow specific properties like the maximum concurrent steps to be scheduled, or changing the default Tuple Comparator class.
`cascading.cascade.CascadeProps`	Allows for setting any Cascade specific properties like the maximum concurrent Flows to be scheduled.
`cascading.tap.hadoop.HfProps`	Allows for setting Hadoop specific FileSystem properties, specifically properties around enabling the 'combined input format' support. Combining inputs minimized the performance penalty around processing large numbers of small files.

4.4 Executing

Running a Cascading application is the same as running any Hadoop application. After packaging your application into a single jar (see Building), you must use `bin/hadoop` to submit the application to the cluster.

For example, to execute an application stuffed into `your-application.jar`, call the Hadoop shell script:

```
$HADOOP_HOME/bin/hadoop jar your-application.jar [some params]
```

Example 4.2 Running a Cascading Application

If the configuration scripts in `$HADOOP_CONF_DIR` are configured to use a cluster, the Jar is pushed into the cluster for execution.

Cascading does not rely on any environment variables like `$HADOOP_HOME` or `$HADOOP_CONF_DIR`, only `bin/hadoop` does.

It should be noted that even though `your-application.jar` is passed on the command line to `bin/hadoop`, this in no way configures Hadoop to push this jar into the cluster. You must still call one of the property setters mentioned above to set the proper path to the application jar. If misconfigured, it's likely that one of the internal libraries (found in the lib folder) will be pushed to the cluster instead, and "Class Not Found" exceptions will be thrown.

4.5 Debugging

Debugging and testing in Cascading local mode, unlike Cascading Hadoop mode, is trivial as all the work and processing happens in the local JVM and in local memory. This dramatically simplifies the use of an IDE and Debugger.Thus the very first recommendation for debugging Cascading applications on Hadoop is to first write tests that run in Cascading local mode.

Along with the use of an IDE Debugger, Cascading provides two tools to help sort out runtime issues. First is the use of the `Debug` filter.

It is a best practice to sprinkle `Debug` operators (see Debug Function) in the pipe assembly and rely on the planner to remove them at runtime by setting a `DebugLevel`. `Debug` can only print to the local console via std out or std error, thus making it harder for use on Hadoop, as Operations do not execute locally but on the cluster side. `Debug` can optionally print the current field names, and a prefix can be set to help distinguish between instances of the `Debug` operation.

Additionally, the actual execution plan for a given Flow can be written out (and visualized) via the Flow.writeDOT() method. DOT files are simply text representation of graph data and can be read by tools like GraphViz and Omni Graffle.

In Cascading local mode, these execution plans are exactly as the pipe assemblies were coded, except the sub-assemblies are unwound and the field names across the Flow are resolved by the local mode planner. That is, `Fields.ALL` and other wild cards are converted the actual field names or ordinals.

In the case of Hadoop mode, using the `HadoopFlowConnector`, the DOT files also contain the intermediate `Tap` instances created to join MapReduce jobs together. Thus the branches between Tap instances are effectively

MapReduce jobs. See the `Flow.writeStepsDOT()` method to write out all the MapReduce jobs that will be scheduled.

This information can also be misleading to what is actually happening per Map or Reduce task cluster side. For a more detailed view of the data pipeline actually executing on a given Map or Reduce task, set the "cascading.stream.dotfile.path" property on the `FlowConnector`. This will write, cluster side, a DOT representation of the current data pipeline path the current Map or Reduce task is handling which is a function of which file(s) the Map or Reduce task are reading and processing. And if multiple files, which files are being read to which `HashJoin` instances. It is recommended to use a relative path like `stepPlan/`.

If the `connect()` method on the current `FlowConnector` fails, the resulting `PlannerException` has a `writeDOT()` method that shows the progress of the current planner.

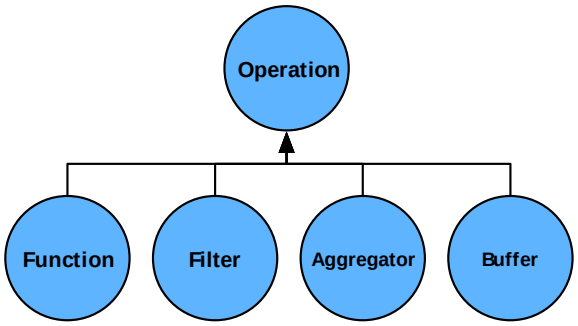
If Cascading is failing with an unknown internal runtime exception during Map or Reduce task startup, setting the "cascading.stream.error.dotfile" property will tell Cascading where to write a DOT representation of the pipeline it was attempting to build, if any. This file will allow the Cascading community to better identify and resolve issues.

5. Using and Developing Operations

5.1 Introduction

So far we've talked about setting up sources and sinks, shaping the data streams, referencing the data fields, and so on. Within this Pipe framework, Operations are used to act upon the data - e.g., alter it, filter it, analyze it, or transform it. You can use the standard Operations in the Cascading library to create powerful and robust applications by combining them in chains (much like Unix operations such as **sed**, **grep**, **sort**, **uniq**, and **awk**). And if you want to go further, it's also very simple to develop custom Operations in Cascading.

There are four kinds of Operations: `Function`, `Filter`, `Aggregator`, and `Buffer`.



Operations typically require an input argument Tuple to act on. And all Operations can return zero or more Tuple object results - except `Filter`, which simply returns a Boolean indicating whether to discard the current Tuple. A `Function`, for instance, can parse a string passed by an argument Tuple and return a new Tuple for every value parsed (i.e., one Tuple for each "word"), or it may create a single Tuple with every parsed value included as an element in one Tuple object (e.g., one Tuple with "first-name" and "last-name" fields).

In theory, a `Function` can be used as a `Filter` by not emitting a Tuple result. However, the `Filter` type is optimized for filtering, and can be combined with logical Operations such as `Not`, `And`, `Or`, etc.

During runtime, Operations actually receive arguments as one or more instances of the `TupleEntry` object. The TupleEntry object holds the current `Tuple` and a `Fields` object that defines field names for positions within the Tuple.

Except for `Filter`, all Operations must declare result Fields, and if the actual output does not match the declaration, the process will fail. For example, consider a `Function` written to parse words out of a String and return a new Tuple for each word. If it declares that its intended output is a Tuple with a single field named "word", and then returns more values in the Tuple beyond that single "word", processing will halt. However, Operations designed to return arbitrary numbers of values in a result Tuple may declare `Fields.UNKNOWN`.

The Cascading planner always attempts to "fail fast" where possible by checking the field name dependencies between Pipes and Operations, but there may be some cases the planner can't account for.

All Operations must be wrapped by either an `Each` or an `Every` pipe instance. The pipe is responsible for passing in an argument Tuple and accepting the resulting output Tuple.

Operations by default are assumed by the Cascading planner to be "safe". A safe Operation is idempotent; it can safely execute multiple times on the exact same record or Tuple; it has no side-effects. If a custom Operation is not idempotent, the method `isSafe()` must return `false`. This value influences how the Cascading planner renders the Flow under certain circumstances.

5.2 Functions

A `Function` expects a stream of individual argument Tuples, and returns zero or more result Tuples for each of them. Like a `Filter`, a `Function` is used with an `Each` pipe, which may follow any pipe type.

To create a custom `Function`, subclass the class `cascading.operation.BaseOperation` and implement the interface `cascading.operation.Function`. Since the `BaseOperation` has been subclassed, the `operate` method, as defined on the `Function` interface, is the only method that must be implemented.

```
public class SomeFunction extends BaseOperation implements Function
  {
  public void operate( FlowProcess flowProcess, FunctionCall functionCall )
    {
    // get the arguments TupleEntry
    TupleEntry arguments = functionCall.getArguments();

    // create a Tuple to hold our result values
    Tuple result = new Tuple();

    // insert some values into the result Tuple

    // return the result Tuple
    functionCall.getOutputCollector().add( result );
    }
  }
```

Example 5.1 Custom Function

Whenever possible, functions should declare both the number of argument values they expect and the field names of the Tuple they return. However, these declarations are optional, as explained below.

For input, functions must accept one or more values in a Tuple as arguments. If not specified, the default is to accept any number of values (`Operation.ANY`). Cascading verifies during planning that the number of arguments selected matches the number of arguments expected.

For output, it's a good practice to declare the field names that a function returns. If not specified, the default is `Fields.UNKNOWN`, meaning that an unknown number of fields are returned in each Tuple.

Both declarations - the number of input arguments and declared result fields - must be done on the constructor, either by passing default values to the `super` constructor, or by accepting the values from the user via a constructor implementation.

```
public class AddValuesFunction extends BaseOperation implements Function
  {
  public AddValuesFunction()
    {
    // expects 2 arguments, fail otherwise
    super( 2, new Fields( "sum" ) );
    }

  public AddValuesFunction( Fields fieldDeclaration )
    {
    // expects 2 arguments, fail otherwise
    super( 2, fieldDeclaration );
    }

  public void operate( FlowProcess flowProcess, FunctionCall functionCall )
    {
    // get the arguments TupleEntry
    TupleEntry arguments = functionCall.getArguments();

    // create a Tuple to hold our result values
    Tuple result = new Tuple();

    // sum the two arguments
    int sum = arguments.getInteger( 0 ) + arguments.getInteger( 1 );

    // add the sum value to the result Tuple
    result.add( sum );

    // return the result Tuple
    functionCall.getOutputCollector().add( result );
    }
  }
```

Example 5.2 Add Values Function

The example above implements a `Function` that accepts two values in the argument Tuple, adds them together, and returns the result in a new Tuple.

The first constructor above assumes a default field name for the field that this `Function` returns. In practice, it's good to give the user the option of overriding the declared field names, allowing them to prevent possible field name collisions that might cause the planner to fail.

This line is especially important:

```
int sum = arguments.getInteger( 0 ) +
          arguments.getInteger( 1 );
```

```
```

Note that ordinal numbers, not field names, are used here to get argument values. If field names had been used, the AddValuesFunction would have been coupled to the incoming stream.

```
public class EfficientAddValuesFunction
  extends BaseOperation<Tuple> implements Function<Tuple>
  {
  public EfficientAddValuesFunction()
    {
    // expects 2 arguments, fail otherwise
    super( 2, new Fields( "sum" ) );
    }

  public EfficientAddValuesFunction( Fields fieldDeclaration )
    {
    // expects 2 arguments, fail otherwise
    super( 2, fieldDeclaration );
    }

  @Override
  public void prepare( FlowProcess flowProcess, OperationCall<Tuple> call )
    {
    // create a reusable Tuple of size 1
    call.setContext( Tuple.size( 1 ) );
    }

  public void operate( FlowProcess flowProcess, FunctionCall<Tuple> call )
    {
    // get the arguments TupleEntry
    TupleEntry arguments = call.getArguments();

    // get our previously created Tuple
    Tuple result = call.getContext();

    // sum the two arguments
    int sum = arguments.getInteger( 0 ) + arguments.getInteger( 1 );

    // set the sum value on the result Tuple
    result.set( 0, sum );

    // return the result Tuple
    call.getOutputCollector().add( result );
    }

  @Override
  public void cleanup( FlowProcess flowProcess, OperationCall<Tuple> call )
    {
    call.setContext( null );
    }
  }
```

Example 5.3 Add Values Function and Context

This example, a minor variation on the previous one, introduces the use of a "context" object and `prepare()` and `cleanup()` methods.

All Operations allow for a context object, simply a user-defined object that holds state between calls to the `operate()` method. This allows for a given instance of the Operation to be thread safe on a platform that may use multiple threads of execution versus multiple processes. It also allows deferring initialization of complex resources until the Operation is engaged.

The `prepare()` and `cleanup()` methods are invoked once per thread of execution, and in the case of the Hadoop platform, only on the cluster side, never on the client.

In the above example, a `Tuple` is used as the context; a more complex type isn't necessary. Also note that the Tuple isn't storing state, but is re-used to reduce the number of new Object instances created. In Cascading, it is perfectly safe to output the same Tuple instance from `operate()`. The method `functionCall.getOutputCollector().add( result )` will not return until the result `Tuple` has been processed or persisted downstream.

5.3 Filter

A `Filter` expects a stream of individual argument Tuples and returns a Boolean value for each one, stating whether it should be discarded. Like a `Function`, a `Filter` is used with an `Each` pipe, which may follow any pipe type.

To create a custom `Filter`, subclass the class `cascading.operation.BaseOperation` and implement the interface `cascading.operation.Filter`. Because `BaseOperation` has been subclassed, the `isRemove` method, as defined on the `Filter` interface, is the only method that must be implemented.

```
public class SomeFilter extends BaseOperation implements Filter
  {
  public boolean isRemove( FlowProcess flowProcess, FilterCall call )
    {
    // get the arguments TupleEntry
    TupleEntry arguments = call.getArguments();

    // initialize the return result
    boolean isRemove = false;

    // test the argument values and set isRemove accordingly

    return isRemove;
    }
  }
```

Example 5.4 Custom Filter

Filters must accept one or more values in a Tuple as arguments, and should declare the number of argument values they expect. If not specified, the default is to accept any number of values (`Operation.ANY`). Cascading verifies during planning that the number of arguments selected matches the number of arguments expected.

The number of arguments declaration must be done on the constructor, either by passing a default value to the `super` constructor, or by accepting the value from the user via a constructor implementation.

```
public class StringLengthFilter extends BaseOperation implements Filter
  {
  public StringLengthFilter()
    {
    // expects 2 arguments, fail otherwise
    super( 2 );
    }

  public boolean isRemove( FlowProcess flowProcess, FilterCall call )
    {
    // get the arguments TupleEntry
    TupleEntry arguments = call.getArguments();

    // filter out the current Tuple if the first argument length is greater
    // than the second argument integer value
    return arguments.getString( 0 ).length() > arguments.getInteger( 1 );
    }
  }
```

Example 5.5 String Length Filter

The example above implements a `Filter` that accepts two arguments and filters out the current Tuple if the first argument, String length, is greater than the integer value of the second argument.

5.4 Aggregator

An `Aggregator` expects a stream of tuple groups (the output of a `GroupBy` or `CoGroup` pipe), and returns zero or more result tuples for every group. An `Aggregator` may only be used with an `Every` pipe - which may follow a `GroupBy`, a `CoGroup`, or another `Every` pipe, but not an `Each`.

To create a custom `Aggregator`, subclass the class `cascading.operation.BaseOperation` and implement the interface `cascading.operation.Aggregator`. Because `BaseOperation` has been subclassed, the `start`, `aggregate`, and `complete` methods, as defined on the `Aggregator` interface, are the only methods that must be implemented.

```
public class SomeAggregator extends BaseOperation<SomeAggregator.Context>
  implements Aggregator<SomeAggregator.Context>
  {
  public static class Context
    {
    Object value;
    }

  public void start( FlowProcess flowProcess,
                     AggregatorCall<Context> aggregatorCall )
    {
    // get the group values for the current grouping
    TupleEntry group = aggregatorCall.getGroup();

    // create a new custom context object
    Context context = new Context();

    // optionally, populate the context object

    // set the context object
    aggregatorCall.setContext( context );
    }

  public void aggregate( FlowProcess flowProcess,
                         AggregatorCall<Context> aggregatorCall )
    {
    // get the current argument values
    TupleEntry arguments = aggregatorCall.getArguments();

    // get the context for this grouping
    Context context = aggregatorCall.getContext();

    // update the context object
    }

  public void complete( FlowProcess flowProcess,
                        AggregatorCall<Context> aggregatorCall )
    {
    Context context = aggregatorCall.getContext();

    // create a Tuple to hold our result values
    Tuple result = new Tuple();

    // insert some values into the result Tuple based on the context

    // return the result Tuple
    aggregatorCall.getOutputCollector().add( result );
    }
  }
```

Example 5.6 Custom Aggregator

Whenever possible, Aggregators should declare both the number of argument values they expect and the field names of the Tuple they return. However, these declarations are optional, as explained below.

For input, Aggregators must accept one or more values in a Tuple as arguments. If not specified, the default is to accept any number of values (`Operation.ANY`). Cascading verifies during planning that the number of arguments selected is the same as the number of arguments expected.

For output, it's good practice for Aggregators to declare the field names they return. If not specified, the default is `Fields.UNKNOWN`, meaning that an unknown number of fields are returned in each Tuple.

Both declarations - the number of input arguments and declared result fields - must be done on the constructor, either by passing default values to the `super` constructor, or by accepting the values from the user via a constructor implementation.

```
public class AddTuplesAggregator
    extends BaseOperation<AddTuplesAggregator.Context>
    implements Aggregator<AddTuplesAggregator.Context>
  {
  public static class Context
    {
    long value = 0;
    }

  public AddTuplesAggregator()
    {
    // expects 1 argument, fail otherwise
    super( 1, new Fields( "sum" ) );
    }

  public AddTuplesAggregator( Fields fieldDeclaration )
    {
    // expects 1 argument, fail otherwise
    super( 1, fieldDeclaration );
    }

  public void start( FlowProcess flowProcess,
                     AggregatorCall<Context> aggregatorCall )
    {
    // set the context object, starting at zero
    aggregatorCall.setContext( new Context() );
    }

  public void aggregate( FlowProcess flowProcess,
                         AggregatorCall<Context> aggregatorCall )
    {
    TupleEntry arguments = aggregatorCall.getArguments();
    Context context = aggregatorCall.getContext();

    // add the current argument value to the current sum
    context.value += arguments.getInteger( 0 );
    }

  public void complete( FlowProcess flowProcess,
                        AggregatorCall<Context> aggregatorCall )
    {
    Context context = aggregatorCall.getContext();

    // create a Tuple to hold our result values
    Tuple result = new Tuple();
```

Example 5.7 Add Tuples Aggregator

```
    // set the sum
    result.add( context.value );

    // return the result Tuple
    aggregatorCall.getOutputCollector().add( result );
```

The example above implements an `Aggregator` that accepts a value in the argument Tuple, adds all the argument tuples in the current grouping, and returns the result as a new Tuple.

The first constructor above assumes a default field name that this `Aggregator` returns. In practice, it's good to give the user the option of overriding the declared field names, allowing them to prevent possible field name collisions that might cause the planner to fail.

There are several constraints on the use of Aggregators that may not be self-evident. These are detailed in the Javadoc

5.5 Buffer

A `Buffer` expects set of argument tuples in the same grouping, and may return zero or more result tuples.

A `Buffer` is very similar to an `Aggregator`, except that it receives the current Grouping Tuple, and an iterator of all the arguments it expects, for every value Tuple in the current grouping - all on the same method call. This is very similar to the typical Reducer interface in MapReduce, and is best used for operations that need visibility to the previous and next elements in the stream - such as smoothing a series of time-stamps where there are missing values.

A `Buffer` may only be used with an `Every` pipe, and it may only follow a `GroupBy` or `CoGroup` pipe type.

To create a custom `Buffer`, subclass the class `cascading.operation.BaseOperation` and implement the interface `cascading.operation.Buffer`. Because `BaseOperation` has been subclassed, the `operate` method, as defined on the `Buffer` interface, is the only method that must be implemented.

```
public class SomeBuffer extends BaseOperation implements Buffer
  {
  public void operate( FlowProcess flowProcess, BufferCall bufferCall )
    {
    // get the group values for the current grouping
    TupleEntry group = bufferCall.getGroup();

    // get all the current argument values for this grouping
    Iterator<TupleEntry> arguments = bufferCall.getArgumentsIterator();

    // create a Tuple to hold our result values
    Tuple result = new Tuple();

    while( arguments.hasNext() )
      {
      TupleEntry argument = arguments.next();

      // insert some values into the result Tuple based on the arguemnts
      }

    // return the result Tuple
    bufferCall.getOutputCollector().add( result );
    }
  }
```

Example 5.8 Custom Buffer

Buffers should declare both the number of argument values they expect and the field names of the Tuple they return.

For input, Buffers must accept one or more values in a Tuple as arguments. If not specified, the default is to accept any number of values (Operation.ANY). During the planning phase, Cascading verifies that the number of arguments selected is the same as the number of arguments expected.

For output, it's good practice for Buffers to declare the field names they return. If not specified, the default is `Fields.UNKNOWN`, meaning that an unknown number of fields are returned in each Tuple.

Both declarations - the number of input arguments and declared result fields - must be done on the constructor, either by passing default values to the `super` constructor, or by accepting the values from the user via a constructor implementation.

```
public class AverageBuffer extends BaseOperation implements Buffer
  {

  public AverageBuffer()
    {
    super( 1, new Fields( "average" ) );
    }

  public AverageBuffer( Fields fieldDeclaration )
    {
    super( 1, fieldDeclaration );
    }

  public void operate( FlowProcess flowProcess, BufferCall bufferCall )
    {
    // init the count and sum
    long count = 0;
    long sum = 0;

    // get all the current argument values for this grouping
    Iterator<TupleEntry> arguments = bufferCall.getArgumentsIterator();

    while( arguments.hasNext() )
      {
      count++;
      sum += arguments.next().getInteger( 0 );
      }

    // create a Tuple to hold our result values
    Tuple result = new Tuple( sum / count );

    // return the result Tuple
    bufferCall.getOutputCollector().add( result );
    }
  }
```

Example 5.9 Average Buffer

The example above implements a buffer that accepts a value in the argument Tuple, adds all these argument tuples in the current grouping, and returns the result divided by the number of argument tuples counted in a new Tuple.

The first constructor above assumes a default field name for the field that this `Buffer` returns. In practice, it's good to give the user the option of overriding the declared field names, allowing them to prevent possible field name collisions that might cause the planner to fail

Note that this example is somewhat artificial. In actual practice, an `Aggregator` would be a better way to compute averages for an entire dataset. A `Buffer` is better suited for calculating running averages across very large spans, for example.

There are several constraints on the use of Buffers that may not be self-evident. These are detailed in the Javadoc.

As with the `Function` example above, a `Buffer` may define a custom context object and implement the `prepare()` and `cleanup()` methods to maintain state, or re-use outgoing `Tuple` instances for efficiency.

5.6 Operation and BaseOperation

In all of the above sections, the `cascading.operation.BaseOperation` class was subclassed. This class is an implementation of the `cascading.operation.Operation` interface, and provides a few default method implementations. It is not strictly required to extend `BaseOperation` when implementing this interface, but it is very convenient to do so.

When developing custom operations, the developer may need to initialize and destroy a resource. For example, when doing pattern matching, you might need to initialize a `java.util.regex.Matcher` and use it in a thread-safe way. Or you might need to open, and eventually close, a remote connection. But for performance reasons, the operation should not create or destroy the connection for each Tuple or every Tuple group that passes through.

For this reason, the interface `Operation` declares two methods: `prepare()` and `cleanup()`. In the case of Hadoop and MapReduce, the `prepare()` and `cleanup()` methods are called once per Map or Reduce task. The `prepare()` method is called before any argument Tuple is passed in, and the `cleanup()` method is called after all Tuple arguments have been operated on. Within each of these methods, the developer can initialize a "context" object that can hold an open socket connection or `Matcher` instance. This context is user defined, and is the same mechanism used by the `Aggregator` operation - except that the `Aggregator` is also given the opportunity to initialize and destroy its context, via the `start()` and `complete()` methods.

Note that if a "context" object is used, its type should be declared in the subclass class declaration using the Java Generics notation.

6. Custom Taps and Schemes

6.1 Introduction

Cascading is designed to be easily configured and enhanced by developers. In addition to creating custom Operations, developers can create custom `Tap` and `Scheme` classes that let applications connect to external systems or read/write data to proprietary formats.

A Tap represents something physical, like a file or a database table. Accordingly, Tap implementations are responsible for life-cycle issues around the resource they represent, such as tests for resource existence, or to perform resource deletion (dropping a remote SQL table).

A Scheme represents a format or representation - such as a text format for a file, the columns in a table, etc. Schemes are used to convert between the source data's native format and a `cascading.tuple.Tuple` instance.

Creating custom taps and schemes can be an involved process. When using the Cascading Hadoop mode, it requires some knowledge of Hadoop and the Hadoop FileSystem API. If a flow needs to support a new file system, passing a fully-qualified URL to the `Hfs` constructor may be sufficient - the `Hfs` tap will look up a file system based on the URL scheme via the Hadoop FileSystem API. If not, a new system is commonly constructed by subclassing the `cascading.tap.Hfs` class.

Delegating to the Hadoop FileSystem API is not a strict requirement. But if not using it, the developer must implement Hadoop `org.apache.hadoop.mapred.InputFormat` and/or `org.apache.hadoop.mapred.OutputFormat` classes so that Hadoop knows how to split and handle the incoming/outgoing data. The custom `Scheme` is responsible for setting the `InputFormat` and `OutputFormat` on the `JobConf`, via the `sinkConfInit` and `sourceConfInit` methods.

For examples of how to implement a custom tap and scheme, see the Cascading Modules [http://cascading.org/modules.html] page.

6.2 Custom Taps

All custom Tap classes must subclass the `cascading.tap.Tap` abstract class and implement the required methods. The method `getIdentifier()` must return a `String` that uniquely identifies the resource the Tap instance is managing. Any two Tap instances with the same fully-qualified identifier value will be considered equal.

Every Tap is presented an opportunity to set any custom properties the underlying platform requires, via the methods `sourceConfInit()` (for a Tuple source tap) and `sinkConfInit()` (for a Tuple sink tap). These two methods may be called more than once with new configuration objects, and should be idempotent.

A Tap is always sourced from the `openForRead()` method via a `TupleEntryIterator` - i.e., `openForRead()` is always called in the same process that will read the data. It is up to the Tap to return a `TupleEntryIterator` that will iterate across the resource, returning a `TupleEntry` instance (and `Tuple` instance) for each "record" in the resource. `TupleEntryIterator.close()` is always called when no more entries will be read. For more on this topic, see `TupleEntrySchemeIterator` in the Javadoc.

On some platforms, `openForRead()` is called with a pre-instantiated Input type. Typically this Input type should be used instead of instantiating a new instance of the appropriate type.

In the case of the Hadoop platform, a `RecordReader` is created by Hadoop and passed to the Tap. This `RecordReader` is already configured to read data from the current `InputSplit`.

Similarly, a Tap is always used to sink data from the `openForWrite()` method via the `TupleEntryCollector`. Here again, `openForWrite()` is always called in the process in which data will be written. It is up to the Tap to return a `TupleEntryCollector` that will accept and store any number of `TupleEntry` or `Tuple` instances for each record that is processed or created by a given Flow. `TupleEntryCollector.close()` is always called when no more entries will be written. See `TupleEntrySchemeCollector` in the Javadoc.

Again, on some platforms, `openForWrite()` will be called with a pre-instantiated Output type. Typically this Output type should be used instead of instantiating a new instance of the appropriate type.

In the case of the Hadoop platform, an `OutputCollector` is created by Hadoop and passed to the Tap. This `OutputCollector` is already configured to to write data to the current resource.

Both the `TupleEntrySchemeIterator` and `TupleEntrySchemeCollector` should be used to hold any state or resources necessary to communicate with any remote services. For example, when connecting to a SQL database, any JDBC drivers should be created on the constructor and cleaned up on `close()`.

Note that the Tap is not responsible for reading or writing data to the Input or Output type. This is delegated to the `Scheme` passed on the constructor of the `Tap`. Consequently, the `Scheme` is responsible for configuring the Input and Output types it will be reading and writing.

6.3 Custom Schemes

All custom Scheme classes must subclass the `cascading.scheme.Scheme` abstract class and implement the required methods.

A `Scheme` is ultimately responsible for sourcing and sinking Tuples of data. Consequently it must know what `Fields` it presents during sourcing, and what `Fields` it accepts during sinking. Thus the constructors on the base `Scheme` type must be set with the source and sink Fields.

A Scheme is allowed to source different Fields than it sinks. The `TextLine Scheme` does just this. (The `TextDelimited Scheme`, on the other hand, forces the source and sink `Fields` to be the same.)

The `retrieveSourceFields()` and `retrieveSinkFields()` methods allow a custom `Scheme` to fetch its source and sink `Fields` immediately before the planner is invoked - for example, from the header of a file, as is the case with `TextDelimited`. Also the `presentSourceFields()` and `presentSinkFields()` methods notify the `Scheme` of the `Fields` that the planner expects the Scheme to handle - for example, to write the field names as a header, as is the case with `TextDelimited`.

Every `Scheme` is presented the opportunity to set any custom properties the underlying platform requires, via the methods `sourceConfInit()` (for a Tuple source tap) and `sinkConfInit()` (for a Tuple sink tap). These methods may be called more than once with new configuration objects, and should be idempotent.

On the Hadoop platform, these methods should be used to configure the appropriate `org.apache.hadoop.mapred.InputFormat` and `org.apache.hadoop.mapred.OutputFormat`.

A Scheme is always sourced via the `source()` method, and is always sunk to via the `sink()` method.

Prior to a `source()` or `sink()` call, the `sourcePrepare()` and `sinkPrepare()` methods are called. After all values have been read or written, the s `ourceCleanup()` and `sinkCleanup()` methods are called.

The `*Prepare()` methods allow a Scheme to initialize any state necessary - for example, to create a new `java.util.regex.Matcher` instance for use against all record reads). Conversely, the `*Cleanup()` methods allow for clearing up any resources.

These methods are always called in the same process space as their associated `source()` and `sink()` calls. In the case of the Hadoop platform, this will likely be on the cluster side, unlike calls to `*ConfInit()` which will likely be on the client side.

7. Field Typing and Type Coercion

7.1 Field Typing

As of Cascading 2.2, the Fields class can hold type information for each field, and the Cascading planner can propagate that information from source Tap instances to downstream Operations through to sink Tap instances.

This allows for Taps to read and store type information for external systems and applications, error detection during joins (detecting non-comparable types), to enforce canonical representations within the Tuple (prevent a field from switching arbitrarily between String and Integer types), and to allow for pluggable coercion from one type to another type, even if either isn't a Java primitive.

To declare types, simply pass type information to the Fields instance either through the constructor or via a fluent API.

```
Fields resultFields = new Fields( "count", Long.class ); // null is ok
```

Example 7.1 Constructor

```
Fields resultFields = new Fields( "count" ).applyTypes( long.class ); // null becomes 0
```

Example 7.2 Fluent

Note the first example uses `Long.class`, and the second `long.class`. Since `Long` is an object, we are letting Cascading know that the null value can be set. If declared `long` (a primitive) then null becomes zero.

In practice, typed fields can only be used when they declare the results of an operation, for example:

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );

// note we do not pass the parent assembly Pipe in
Fields valueField = new Fields( "size" );
Fields sumField = new Fields( "total-size", long.class );
SumBy sumBy = new SumBy( valueField, sumField );

Fields countField = new Fields( "num-events" );
CountBy countBy = new CountBy( countField );

assembly = new AggregateBy( assembly, groupingFields, sumBy, countBy );
```

Example 7.3 Declaring Typed Results

Here the type information serves two roles. First, it allows a downstream consumer of the field value to know the type maintained in the tuple. Second, the SumBy sub-assembly now has a simpler API and can get the type information it needs internally to perform the aggregation directly from the Fields instance.

Note that the `TextDelimited` and other `Scheme` classes should have any type information declared so it can be maintained by the Cascading planner. Custom `Scheme` types also have the opportunity to read type information from any field or data sources they represent so it can be handed to the planner during runtime.

7.2 Type Coercion

Type coercion is a means to convert one data type to another. For example, parsing the Java `String` "42" to the `Integer` 42 would be coercion. Or more simply, converting a `Long` 42 to a `Double` 42.0. Cascading supports primitive type coercions natively through the `cascading.tuple.coerce.Coercions` class.

In practice, developers implicitly invoke coercions via the `cascading.tuple.TupleEntry` interface by requesting a `Long` or `String` representation of a field, via `TupleEntry.getLong()` or `TupleEntry.getString()`, respectively.

Or when data is set on a `Tuple` via `TupleEntry.setLong()` or `TupleEntry.setString()`, for example. If the field was declared as an `Integer`, and `TupleEntry.setString( "someField", "42" )` was called, the value of "someFields" will be coerced into its canonical form, 42.

To create custom coercions, the `cascading.tuple.type.CoercibleType` interface must be implemented, and instances of `CoercibleType` can be used as the Type accepted by the Fields API as `CoercibleType` extends `java.lang.reflect.Type`.

Cascading provided a `cascading.tuple.type.DateType` implementation to allow for coercions between date strings and the `Long` canonical type. For example:

```
SimpleDateFormat dateFormat = new SimpleDateFormat( "dd/MMM/yyyy:HH:mm:ss:SSS Z" );
Date firstDate = new Date();
String stringFirstDate = dateFormat.format( firstDate );

CoercibleType coercible = new DateType( "dd/MMM/yyyy:HH:mm:ss:SSS Z", TimeZone.getDefault

// create the Fields, Tuple, and TupleEntry
Fields fields = new Fields( "dateString", "dateValue" ).applyTypes( coercible, long.class
Tuple tuple = new Tuple( firstDate.getTime(), firstDate.getTime() );
TupleEntry results = new TupleEntry( fields, tuple );

// test the results
assert results.getObject( "dateString" ).equals( firstDate.getTime() );
assert results.getLong( "dateString" ) == firstDate.getTime();
assert results.getString( "dateString" ).equals( stringFirstDate );
assert !results.getString( "dateString" ).equals( results.getString( "dateValue" ) ); //

Date secondDate = new Date( firstDate.getTime() + ( 60 * 1000 ) );
String stringSecondDate = dateFormat.format( secondDate );

results.setString( "dateString", stringSecondDate );
results.setLong( "dateValue", secondDate.getTime() );

assert !results.getObject( "dateString" ).equals( firstDate.getTime() ); // equals
assert results.getObject( "dateString" ).equals( secondDate.getTime() ); // not equals
```

Example 7.4 Date Type

In this example we declare the "dateString" field to be a `DateType`. `DateType` maintains the value of the field as a `long` internally, but if a `String` is set or requested, it will be converted using the given `SimpleDateFormat String` against the given `TimeZone`. In the case of a `TextDelimited` CSV file, where one column is a date value, `DateType` can be used to declare its format allowing `TextDelimited` to read and write the value as a `String`, but use the value internally (in the Tuple) as a `long`, which is much more efficient.

8. Advanced Processing

8.1 SubAssemblies

In Cascading, SubAssemblies are reusable pipe assemblies that are linked into larger pipe assemblies. They function much like subroutines in a larger program. SubAssemblies are a good way to organize complex pipe assemblies, and they allow for commonly-used pipe assemblies to be packaged into libraries for inclusion in other projects by other users.

To create a SubAssembly, subclass the `cascading.pipe.SubAssembly` class.

```
public class SomeSubAssembly extends SubAssembly
  {
  public SomeSubAssembly( Pipe lhs, Pipe rhs )
    {
    // must register incoming pipes
    setPrevious( lhs, rhs );

    // continue assembling against lhs
    lhs = new Each( lhs, new SomeFunction() );
    lhs = new Each( lhs, new SomeFilter() );

    // continue assembling against rhs
    rhs = new Each( rhs, new SomeFunction() );

    // joins the lhs and rhs
    Pipe join = new CoGroup( lhs, rhs );

    join = new Every( join, new SomeAggregator() );

    join = new GroupBy( join );

    join = new Every( join, new SomeAggregator() );

    // the tail of the assembly
    join = new Each( join, new SomeFunction() );

    // must register all assembly tails
    setTails( join );
    }
  }
```

Example 8.1 Creating a SubAssembly

In the example above, we pass in (as parameters via the constructor) the pipes that we wish to continue assembling against, in the first line we register the incoming "previous" pipes, and in the last line we register the outgoing "join" pipe as a tail. This allows SubAssemblies to be nested within larger pipe assemblies or other SubAssemblies.

```
// the "left hand side" assembly head
Pipe lhs = new Pipe( "lhs" );

// the "right hand side" assembly head
Pipe rhs = new Pipe( "rhs" );

// our custom SubAssembly
Pipe pipe = new SomeSubAssembly( lhs, rhs );

pipe = new Each( pipe, new SomeFunction() );
```

Example 8.2 Using a SubAssembly

The example above demonstrates how to include a SubAssembly into a new pipe assembly.

Note that in a SubAssembly that represents a split - that is, a SubAssembly with two or more tails - you can use the `getTails()` method to access the array of tails set internally by the `setTails()` method.

```
public class SplitSubAssembly extends SubAssembly
  {
  public SplitSubAssembly( Pipe pipe )
    {
    // must register incoming pipe
    setPrevious( pipe );

    // continue assembling against pipe
    pipe = new Each( pipe, new SomeFunction() );

    Pipe lhs = new Pipe( "lhs", pipe );
    lhs = new Each( lhs, new SomeFunction() );

    Pipe rhs = new Pipe( "rhs", pipe );
    rhs = new Each( rhs, new SomeFunction() );

    // must register all assembly tails
    setTails( lhs, rhs );
    }
  }
```

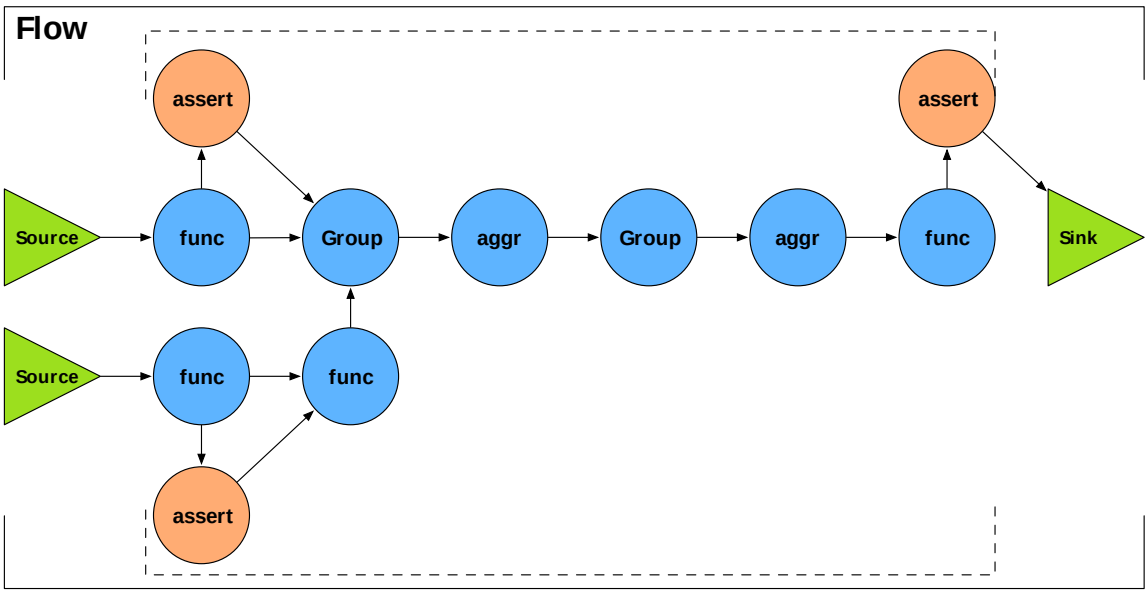
Example 8.3 Creating a Split SubAssembly

```
// the "left hand side" assembly head
Pipe head = new Pipe( "head" );

// our custom SubAssembly
SubAssembly pipe = new SplitSubAssembly( head );

// grab the split branches
Pipe lhs = new Each( pipe.getTails()[ 0 ], new SomeFunction() );
Pipe rhs = new Each( pipe.getTails()[ 1 ], new SomeFunction() );
```

Example 8.4 Using a Split SubAssembly

To rephrase, if a `SubAssembly` does not split the incoming Tuple stream, the SubAssembly instance can be passed directly to the next Pipe instance. But, if the `SubAssembly` splits the stream into multiple branches, handles will be needed to access them. The solution is to pass each branch tail to the `setTails()` method, and call the `getTails()` method to get handles for the desired branches, which can be passed to subsequent instances of `Pipe`.

8.2 Stream Assertions



Stream assertions are simply a mechanism for asserting that one or more values in a tuple stream meet certain criteria. This is similar to the Java language "assert" keyword, or a unit test. An example would be "assert not null" or "assert matches".

Assertions are treated like any other function or aggregator in Cascading. They are embedded directly into the pipe assembly by the developer. By default, if an assertion fails, the processing fails. As an alternative, an assertion failure can be caught by a failure Trap.

Assertions may be more, or less, desirable in different contexts. For this reason, stream assertions can be treated as either "strict" or "validating". *Strict* assertions make sense when running tests against regression data - which should be small, and should represent many of the edge cases that the processing assembly must robustly support. *Validating*

assertions, on the other hand, make more sense when running tests in staging, or when using data that may vary in quality due to an unmanaged source.

And of course there are cases where assertions are unnecessary and only impede processing, and it would be best to just bypass them altogether.

To handle all three of these situations, Cascading can be instructed to *plan out* (i.e., omit) strict assertions, validation assertions, or both when building the Flow. To create optimal performance, Cascading implements this by actually leaving the undesired assertions out of the final Flow (not merely switching them off).

```
// incoming -> "ip", "time", "method", "event", "status", "size"

AssertNotNull notNull = new AssertNotNull();
assembly = new Each( assembly, AssertionLevel.STRICT, notNull );

AssertSizeEquals equals = new AssertSizeEquals( 6 );
assembly = new Each( assembly, AssertionLevel.STRICT, equals );

AssertMatchesAll matchesAll = new AssertMatchesAll( "(GET|HEAD|POST)" );
assembly = new Each( assembly, new Fields( "method" ),
  AssertionLevel.STRICT, matchesAll );

// outgoing -> "ip", "time", "method", "event", "status", "size"
```

Example 8.5 Adding Assertions

Again, assertions are added to a pipe assembly like any other operation, except that the `AssertionLevel` must be set to tell the planner how to treat the assertion during planning.

```
// FlowDef is a fluent way to define a Flow
FlowDef flowDef = new FlowDef();

// bind the taps and pipes
flowDef
  .addSource( assembly.getName(), source )
  .addSink( assembly.getName(), sink )
  .addTail( assembly );

// removes all assertions from the Flow
flowDef
  .setAssertionLevel( AssertionLevel.NONE );

Flow flow = new HadoopFlowConnector().connect( flowDef );
```

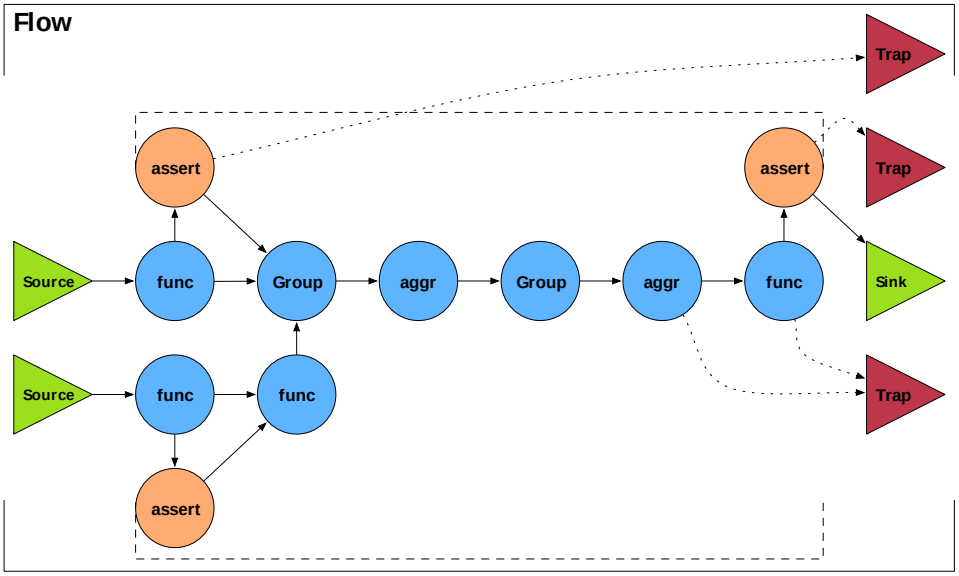
Example 8.6 Planning Out Assertions

To configure the planner to remove some or all assertions, a property can be set via the `FlowConnectorProps.setAssertionLevel()` method or directly on the `FlowDef` instance, as shown

above. Setting `AssertionLevel.NONE` removes all assertions. `AssertionLevel.VALID` keeps `VALID` assertions but removes `STRICT` ones. And `AssertionLevel.STRICT` keeps all assertions - the planner default value.

8.3 Failure Traps

The following diagram shows the use of *Failure Traps* in a pipe assembly.



Failure Traps are similar to tap sinks (as opposed to tap sources) in that they allow data to be stored. The difference is that Tap sinks are bound to a particular tail pipe in a pipe assembly and are the primary outlet of a branch in a pipe assembly. Traps can be bound to intermediate pipe assembly branches - just like Stream Assertions - yet they only capture data that causes an Operation to fail (throw an Exception).

Whenever an operation fails and throws an exception, if there is an associated trap, the offending Tuple is saved to the resource specified by the trap Tap. This allows the job to continue processing without any data loss.

By design, clusters are hardware fault-tolerant - lose a node, and the cluster continues working. But fault tolerance for software is a little different. Failure Traps provide a means for the processing to continue without losing track of the data that caused the fault. For high fidelity applications, this may not be very useful, since you likely will want any errors during processing to cause the application to stop. But for low fidelity applications such as webpage indexing, where skipping a page or two out of a few million is acceptable, this can dramatically improve processing reliability.

```
// ...some useful pipes here

// name this pipe assembly segment
assembly = new Pipe( "assertions", assembly );

AssertNotNull notNull = new AssertNotNull();
assembly = new Each( assembly, AssertionLevel.STRICT, notNull );

AssertSizeEquals equals = new AssertSizeEquals( 6 );
assembly = new Each( assembly, AssertionLevel.STRICT, equals );

AssertMatchesAll matchesAll = new AssertMatchesAll( "(GET|HEAD|POST)" );
Fields method = new Fields( "method" );
assembly =
  new Each( assembly, method, AssertionLevel.STRICT, matchesAll );

// ...some more useful pipes here

FlowDef flowDef = new FlowDef();

flowDef
  .setName( "log-parser" )
  .addSource( "logs", source )
  .addTailSink( assembly, sink );

// set the trap on the "assertions" branch
flowDef
  .addTrap( "assertions", trap );

FlowConnector flowConnector = new HadoopFlowConnector();
Flow flow =
  flowConnector.connect( flowDef );
```

Example 8.7 Setting Traps

The example above binds a trap tap to the pipe assembly segment named "assertions". Note how we can name branches and segments by using a single `Pipe` instance, and that the naming applies to all subsequent `Pipe` instances.

Traps are for exceptional cases, in the same way that Java Exception handling is. Traps are not intended for application flow control, and not a means to filter some data into other locations. Applications that need to filter out bad data should do so explicitly, using filters. For more on this, see Handling Good and Bad Data.

8.4 Checkpointing

New to Cascading 2, and only supported by the Hadoop planner, is the ability to "checkpoint" data within a Flow by using the `cascading.pipe.Checkpoint Pipe`. That is, a Tuple stream can be persisted to disk at most any arbitrary point. Doing so forces a new FlowStep (MapReduce job when using Hadoop) after the checkpoint position.

By default the checkpoint is anonymous and is cleaned up immediately after the Flow completes. This feature is useful when used in conjunction with a HashJoin where the small side of the join starts out extremely large but is filtered down to fit into memory before being read into the HashJoin. By forcing a checkpoint before the HashJoin, only the small filtered version of the data is replicated over the cluster. Without the checkpoint, it is likely the full unfiltered file will be replicated to every node the FlowStep is executing.

Alternatively, checkpointing is useful for debugging when used with a checkpoint Tap, where the Tap has specified a TextDelimited Scheme without any declared Fields.

```
// the "left hand side" assembly head
Pipe lhs = new Pipe( "lhs" );

lhs = new Each( lhs, new SomeFunction() );
lhs = new Each( lhs, new SomeFilter() );

// the "right hand side" assembly head
Pipe rhs = new Pipe( "rhs" );

rhs = new Each( rhs, new SomeFunction() );

// joins the lhs and rhs
Pipe join = new CoGroup( lhs, rhs );

join = new Every( join, new SomeAggregator() );

// we want to see the data passing through this point
Checkpoint checkpoint = new Checkpoint( "checkpoint", join );

Pipe groupBy = new GroupBy( checkpoint );

groupBy = new Every( groupBy, new SomeAggregator() );

// the tail of the assembly
groupBy = new Each( groupBy, new SomeFunction() );

Tap lhsSource = new Hfs( new TextLine(), "lhs.txt" );
Tap rhsSource = new Hfs( new TextLine(), "rhs.txt" );

Tap sink = new Hfs( new TextLine(), "output" );

// write all data as a tab delimited file, with headers
Tap checkpointTap =
  new Hfs( new TextDelimited( true, "\t" ), "checkpoint" );

FlowDef flowDef = new FlowDef()
  .setName( "flow-name" )
  .addSource( rhs, rhsSource )
  .addSource( lhs, lhsSource )
  .addTailSink( groupBy, sink )
  .addCheckpoint( checkpoint, checkpointTap ); // bind the checkpoint tap

Flow flow = new HadoopFlowConnector().connect( flowDef );
```

Example 8.8 Adding a Checkpoint

As can be seen above, we instantiate a new `Checkpoint` tap by passing it the previous `Every` `Pipe`. This will be the point at which data is persisted. Since we wish to keep the data after the `Flow` has completed, we create a `checkpointTap` that saves the data as a TAB delimited text file. We also specify that field names should be written out into a header file on the `TextDelimited` constructor. Finally the `Tap` is bound to the `Checkpoint` `Pipe` using the `FlowDef`.

8.5 Restarting a Checkpointed Flow

When using Checkpoint pipes in a Flow and the Flow fails, a future execution of the Flow can be restarted after the last successful FlowStep writing to a checkpoint file. That is, a Flow will only restart from the last Checkpoint Pipe location.

This feature requires that the failed Flow be planned with a `runID` set on the FlowDef, and the retry Flow use the same `runID` value. It goes without saying, the retry Flow should be (roughly) equivant to the previous failed attempt.

```
FlowDef flowDef = new FlowDef()
  .setName( "flow-name" )
  .addSource( rhs, rhsSource )
  .addSource( lhs, lhsSource )
  .addTailSink( groupBy, sink )
  .addCheckpoint( checkpoint, checkpointTap )
  .setRunID( "some-unique-value" ); // re-use this id to restart this flow

Flow flow = new HadoopFlowConnector().connect( flowDef );
```

Example 8.9 Setting runID

Caution should be used when using restarted checkpoint Flows. If the input data has changed, or the pipe assembly has significantly been altered, the Flow may fail or there may be undetectable errors.

Note that when using a `runID`, all Flow instances must use a unique value unless they are intended as a retry attempt. The runID value is used to scope the directories for the temporary checkpoint files to prevent file name collisions.

On successful completion of a Flow with a runID, all temporary checkpoint files will be removed, if any.

8.6 Flow and Cascade Event Handling

Each Flow and Cascade has the ability to execute callbacks via an event listener. This ability is useful when an external application needs to be notified that either a Flow or Cascade has started, halted, completed, or either has thrown an exception.

For instance, at the completion of a flow that runs on an Amazon EC2 Hadoop cluster, an Amazon SQS message can be sent to notify another application to fetch the job results from S3 or begin the shutdown of the cluster.

Flows support event listeners through the `cascading.flow.FlowListener` interface and Cascades support event listeners through the `cascading.cascade.CascadeListener`, which supports four events:

onStarting
: The onStarting event is fired when a Flow or Cascade instance receives the `start()` message.

onStopping
: The onStopping event is fired when a Flow or Cascade instance receives the `stop()` message.

onCompleted
: The onCompleted event is fired when a Flow or Cascade instance has completed all work, regardless of success or failure. If an exception was thrown, onThrowable will be fired before this event.

onThrowable
: The onThrowable event is fired if any internal job client throws a Throwable type. This throwable is passed as an argument to the event. onThrowable should return true if the given throwable was handled, and should not be rethrown from the `Flow.complete()` or `Cascade.complete()` methods.

8.7 PartitionTaps

The `PartitionTap` `Tap` class provides a simple means to break large datasets into smaller sets based on data item values. This is also commonly called *binning* the data, where each "bin" of data is named after some data value(s) shared by the members of that bin. For example, this is a simple way to organize log files by month and year. PartitionTap replaces the TemplateTap in previous versions of Cascading and adds the ability for a PartitionTap instance to be used as both a sink and a source. Previously, TemplateTap could only be used as a sink.

```
TextDelimited scheme =
  new TextDelimited( new Fields( "entry" ), "\t" );
Hfs parentTap = new Hfs( scheme, path );

// dirs named "[year]-[month]"
DelimitedPartition partition = new DelimitedPartition( new Fields( "year", "month" ), "-"
Tap monthsTap = new PartitionTap( parentTap, partition, SinkMode.REPLACE );
```

In the example above, we construct a parent `Hfs tap` and pass it to the constructor of a `PartitionTap` instance, along with a `cascading.tap.partition.DelimitedPartition` "partitioner". If more complex path formatting is necessary, you may implement the `cascading.tap.partition.Partition` interface.

It is important to see in the above example that the `parentTap` will only sink "entry" fields to a text delimited file. But the `monthsTap` expects "year", "month", and "entry" fields from the tuple stream. Here data is stored in the directory name for each partition when the PartitionTap is a sink, there is no need to redundantly store the data in the text delimited file. When reading from a `PartitionTap`, the directory name will be parsed and its values will be added to the outgoing tuple stream when the `PartitionTap` is a source.

Note that you can only create sub-directories to bin data into. Hadoop must still write "part" files into each bin directory, and there is no safe mechanism for manipulating part file names.

One last thing to keep in mind is whether binning happens during the Map phase or the Reduce phase. By doing a `GroupBy` on the values used to populate the template, binning will happen during the Reduce phase, and will likely scale much better in cases where there are a very large number of unique values used in the template resulting in a large number of directories.

8.8 Partial Aggregation instead of Combiners

In Hadoop mode, Cascading does not support MapReduce "Combiners". Combiners are a simple optimization allowing some Reduce functions to run on the Map side of MapReduce. Combiners are very powerful in that they reduce the I/O between the Mappers and Reducers - why send all of your Mapper data to Reducers when you can compute some values on the Map side and combine them in the Reducer? But Combiners are limited to Associative and Commutative functions only, such as "sum" and "max". And the process requires that the values emitted by the Map task must be serialized, sorted (which involves deserialization and comparison), deserialized again, and operated on - after which the results are again serialized and sorted. Combiners trade CPU for gains in I/O.

Cascading takes a different approach. It provides a mechanism to perform partial aggregations on the Map side and combine the results on the Reduce side, but trades memory, instead of CPU, for I/O gains by caching values (up to a threshold limit). This bypasses the redundant serialization, deserialization, and sorting. Also, Cascading allows any aggregate function to be implemented - not just Associative and Commutative functions.

Cascading supports a few built-in partial aggregate operations, including AverageBy, CountBy, SumBy, and FirstBy. These are actually SubAssemblies, not Operations, and are subclasses of the AggregateBy SubAssembly. For more on this, see the section on AggregateBy.

Using partial aggregate operations is quite easy. They are actually less verbose than a standard Aggregate operation.

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );
Fields valueField = new Fields( "size" );
Fields sumField = new Fields( "total-size" );
assembly =
  new SumBy( assembly, groupingFields, valueField, sumField, long.class );
```

Example 8.10 Using a SumBy

For composing multiple partial aggregate operations, things are done a little differently.

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );

// note we do not pass the parent assembly Pipe in
Fields valueField = new Fields( "size" );
Fields sumField = new Fields( "total-size", long.class );
SumBy sumBy = new SumBy( valueField, sumField );

Fields countField = new Fields( "num-events" );
CountBy countBy = new CountBy( countField );

assembly = new AggregateBy( assembly, groupingFields, sumBy, countBy );
```

Example 8.11 Composing partials with AggregateBy

It's important to note that a `GroupBy` Pipe is embedded in the resulting assemblies above. But only one GroupBy is performed in the case of the AggregateBy, and all of the partial aggregations will be performed simultaneously. It is also important to note that, depending on the final pipe assembly, the Map side partial aggregate functions may be planned into the previous Reduce operation in Hadoop, further improving the performance of the application.

9. Built-In Operations

9.1 Identity Function

The `cascading.operation.Identity` function is used to "shape" a tuple stream. Here are some common patterns that illustrate how Cascading "field algebra" works. (Note that, in actual practice, some of these example tasks might be better performed with helper subassemblies such as `cascading.pipe.assembly.Rename`, `cascading.pipe.assembly.Retain`, and `cascading.pipe.assembly.Discard`.)

Discard unused fields

Here Identity passes its arguments out as results, thanks to the `Fields.ARGS` field declaration.

```
// incoming -> "ip", "time", "method", "event", "status", "size"

Identity identity = new Identity( Fields.ARGS );
Fields ipMethod = new Fields( "ip", "method" );
pipe =
  new Each( pipe, ipMethod, identity, Fields.RESULTS );

// outgoing -> "ip", "method"
```

In practice the field declaration can be left out, as `Field.ARGS` is the default declaration for the Identity function. And `Fields.RESULTs` can be left off, as it is the default for the `Every` pipe. Thus, simpler code yields the same result:

```
// incoming -> "ip", "time", "method", "event", "status", "size"

pipe = new Each( pipe, new Fields( "ip", "method" ), new Identity() );

// outgoing -> "ip", "method"
```

Rename all fields

Here Identity renames the incoming arguments. Since Fields.RESULTS is implied, the incoming Tuple is replaced by the selected arguments and given new field names as declared on Identity.

```
// incoming -> "ip", "method"

Identity identity = new Identity( new Fields( "address", "request" ) );
pipe = new Each( pipe, new Fields( "ip", "method" ), identity );

// outgoing -> "address", "request"
```

In the example above, if there were more fields than "ip" and "method", it would work fine - all the extra fields would be discarded. But if the same were true for the next example, the planner would fail.

```
// incoming -> "ip", "method"

Identity identity = new Identity( new Fields( "address", "request" ) );
pipe = new Each( pipe, Fields.ALL, identity );

// outgoing -> "address", "request"
```

Since `Fields.ALL` is the default argument selector for the `Each` pipe, it can be left out as shown below. Again, the above and below examples will fail unless there are exactly two fields in the tuples of the incoming stream.

```
// incoming -> "ip", "method"

Identity identity = new Identity( new Fields( "address", "request" ) );
pipe = new Each( pipe, identity );

// outgoing -> "address", "request"
```

Rename a single field

Here we rename a single field and return it, along with an input Tuple field, as the result. All other fields are dropped.

```
// incoming -> "ip", "time", "method", "event", "status", "size"

Fields fieldSelector = new Fields( "address", "method" );
Identity identity = new Identity( new Fields( "address" ) );
pipe = new Each( pipe, new Fields( "ip" ), identity, fieldSelector );

// outgoing -> "address", "method"
```

Coerce values to specific primitive types

Here we replace the Tuple String values "status" and "size" with `int` and `long` values, respectively. All other fields are dropped.

```
// incoming -> "ip", "time", "method", "event", "status", "size"

Identity identity = new Identity( Integer.TYPE, Long.TYPE );
pipe = new Each( pipe, new Fields( "status", "size" ), identity );

// outgoing -> "status", "size"
```

Or we can replace just the Tuple String value "status" with an `int`, while keeping all the other values in the output Tuple.

```
// incoming -> "ip", "time", "method", "event", "status", "size"
```

```
Identity identity = new Identity( Integer.TYPE );
pipe =
  new Each( pipe, new Fields( "status" ), identity, Fields.REPLACE );

// outgoing -> "ip", "time", "method", "event", "status", "size"
```

9.2 Debug Function

The `cascading.operation.Debug` function is a utility function (actually, it's a `Filter`) that prints the current argument Tuple to either `stdout` or `stderr`. Used with one of the `DebugLevel` enum values (`NONE`, `DEFAULT`, or `VERBOSE`), different debug levels can be embedded in a pipe assembly.

The example below inserts a `Debug` operation at the `VERBOSE` level, but configures the planner to remove all `Debug` operations from the resulting `Flow`.

```
Pipe assembly = new Pipe( "assembly" );

// ...
assembly = new Each( assembly, DebugLevel.VERBOSE, new Debug() );
// ...

// head and tail have same name
FlowDef flowDef = new FlowDef()
  .setName( "debug" )
  .addSource( "assembly", source )
  .addSink( "assembly", sink )
  .addTail( assembly );


// tell the planner to remove all Debug operations
flowDef
  .setDebugLevel( DebugLevel.NONE );

// ...
FlowConnector flowConnector = new HadoopFlowConnector();

Flow flow = flowConnector.connect( flowDef );
```

Note that if the above Flow is run on a cluster, the `stdout` on the cluster nodes will be used. Nothing from the debug output will display on the client side. Debug is only useful when testing things in an IDE or if the remote logs are readily available.

9.3 Sample and Limit Functions

The Sample and Limit functions are used to limit the number of tuples that pass through a pipe assembly.

Sample
: The `cascading.operation.filter.Sample` filter allows a percentage of tuples to pass.

Limit
: The `cascading.operation.filter.Limit` filter allows a set number of tuples to pass.

9.4 Insert Function

The `cascading.operation.Insert` function allows for insertion of constant literal values into the tuple stream.

This is most useful when a splitting a tuple stream and one of the branches needs some identifying value, or when some missing parameter or value, like a date String for the current date, needs to be inserted.

9.5 Text Functions

Cascading includes a number of text functions in the `cascading.operation.text` package.

DateFormatter
: The `cascading.operation.text.DateFormatter` function is used to convert a date timestamp to a formatted String. This function expects a `long` value representing the number of milliseconds since January 1, 1970, 00:00:00 GMT/UTC, and formats the output using `java.text.SimpleDateFormat` syntax.

```
// "ts" -> 1188604863000

DateFormatter formatter =
  new DateFormatter( new Fields( "date" ), "dd/MMM/yyyy" );
pipe = new Each( pipe, new Fields( "ts" ), formatter );

// outgoing -> "date" -> 31/Aug/2007
```

: The example above converts a `long` timestamp ("ts") to a date String.

DateParser
: The `cascading.operation.text.DateParser` function is used to convert a text date String to a timestamp, using the `java.text.SimpleDateFormat` syntax. The timestamp is a `long` value representing the number of milliseconds since January 1, 1970, 00:00:00 GMT/UTC. By default, the output is a field with the name "ts" (for timestamp), but this can be overridden by passing a declared Fields value.

```
// "time" -> 01/Sep/2007:00:01:03 +0000

DateParser dateParser = new DateParser( "dd/MMM/yyyy:HH:mm:ss Z" );
pipe = new Each( pipe, new Fields( "time" ), dateParser );

// outgoing -> "ts" -> 1188604863000
```

: In the example above, an Apache log-style date-time field is converted into a `long` timestamp in UTC.

FieldJoiner

The `cascading.operation.text.FieldJoiner` function joins all the values in a Tuple with a specified delimiter and places the result into a new field. (For the opposite effect, see the `RegexSplitter` function.)

FieldFormatter

The `cascading.operation.text.FieldFormatter` function formats Tuple values with a given String format and stuffs the result into a new field. The `java.util.Formatter` class is used to create a new formatted String.

9.6 Regular Expression Operations

RegexSplitter

The `cascading.operation.regex.RegexSplitter` function splits an argument value based on a regex pattern String. (For the opposite effect, see the FieldJoiner function.) Internally, this function uses `java.util.regex.Pattern.split()`, and it behaves accordingly. By default, it splits on the TAB character ("\t"). If it is known that a determinate number of values will emerge from this function, it can declare field names. In this case, if the splitter encounters more split values than field names, the remaining values are discarded. For more information, see `java.util.regex.Pattern.split( input, limit )`.

RegexParser

The `cascading.operation.regex.RegexParser` function is used to extract a regex-matched value from an incoming argument value. If the regular expression is sufficiently complex, an `int` array may be provided to specify which regex groups should be returned in which field names.

```
// incoming -> "line"

String regex =
  "^([^ ]*) +[^ ]* +[^ ]* +\\[([^]]*)\\] +" +
    "\\\"([^ ]*) ([^ ]*) [^ ]*\\\" ([^ ]*) ([^ ]*).*$";
Fields fieldDeclaration =
  new Fields( "ip", "time", "method", "event", "status", "size" );
int[] groups = {1, 2, 3, 4, 5, 6};
RegexParser parser = new RegexParser( fieldDeclaration, regex, groups );
assembly = new Each( assembly, new Fields( "line" ), parser );

// outgoing -> "ip", "time", "method", "event", "status", "size"
```

In the example above, a line from an Apache access log is parsed into its component parts. Note that the `int[]` groups array starts at 1, not 0. Group 0 is the whole group, so if the first field is included, it is a copy of "line" and not "ip".

RegexReplace

The `cascading.operation.regex.RegexReplace` function is used to replace a regex-matched value with a specified replacement value. It can operate in a "replace all" or "replace first" mode. For more information, see the methods `java.util.regex.Matcher.replaceAll()` and `java.util.regex.Matcher.replaceFirst()`.

```
// incoming -> "line"

RegexReplace replace =
  new RegexReplace( new Fields( "clean-line" ), "\\s+", " ", true );
assembly = new Each( assembly, new Fields( "line" ), replace );

// outgoing -> "clean-line"
```

In the example above, all adjoined white space characters are replaced with a single space character.

RegexFilter

The `cascading.operation.regex.RegexFilter` function filters a Tuple stream based on a specified regex value. By default, tuples that match the given pattern are kept, and tuples that do not match are filtered out. This can be reversed by setting "removeMatch" to `true`. Also, by default, the whole Tuple is matched against the given regex String (in tab-delimited sections). If "matchEachElement" is set to `true`, the pattern is applied to each Tuple value individually. For more information, see the `java.util.regex.Matcher.find()` method.

```
// incoming -> "ip", "time", "method", "event", "status", "size"

Filter filter = new RegexFilter( "^68\\..*" );
assembly = new Each( assembly, new Fields( "ip" ), filter );

// outgoing -> "ip", "time", "method", "event", "status", "size"
```

The above keeps all lines in which "68." appears at the start of the IP address.

RegexGenerator

The `cascading.operation.regex.RegexGenerator` function emits a new tuple for every string (found in an input tuple) that matches a specified regex pattern.

```
// incoming -> "line"

String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
Function function = new RegexGenerator( new Fields( "word" ), regex );
assembly = new Each( assembly, new Fields( "line" ), function );

// outgoing -> "word"
```

Above each "line" in a document is parsed into unique words and stored in the "word" field of each result Tuple.

RegexSplitGenerator

The `cascading.operation.regex.RegexSplitGenerator` function emits a new Tuple for every split on the incoming argument value delimited by the given pattern String. The behavior is similar to the `RegexSplitter` function, except that (assuming multiple matches) `RegexSplitter` emits a single tuple that may contain multiple values, and `RegexSplitGenerator` emits multiple tuples that each contain only one value, as does `RegexGenerator`.

9.7 Java Expression Operations

Cascading provides some support for dynamically-compiled Java expressions to be used in either `Functions` or `Filters`. This capability is provided by the Janino embedded Java compiler, which compiles the expressions into byte code for optimal processing speed. Janino is documented in detail on its website, http://www.janino.net/.

This capability allows an Operation to evaluate a suitable one-line Java expression, such as `a + 3 * 2` or `a < 7`, where the variable values (`a` and `b`) are passed in as Tuple fields. The result of the Operation thus depends on the evaluated result of the expression - in the first example, some number, and in the second, a Boolean value.

ExpressionFunction

The function `cascading.operation.expression.ExpressionFunction` dynamically composes a string expression when executed, assigning argument Tuple values to variables in the expression.

```
// incoming -> "ip", "time", "method", "event", "status", "size"

String exp =
  "\"this \" + method + \" request was \" + size + \" bytes\"";
Fields fields = new Fields( "pretty" );
ExpressionFunction function =
  new ExpressionFunction( fields, exp, String.class );

assembly =
  new Each( assembly, new Fields( "method", "size" ), function );

// outgoing -> "pretty" = "this GET request was 1282652 bytes"
```

Above, we create a new String value that contains an expression containing values from the current Tuple. Note that you must declare the type for every input Tuple field so that the expression compiler knows how to treat the variables in the expression.

ExpressionFilter

The filter `cascading.operation.expression.ExpressionFilter` evaluates a Boolean expression, assigning argument Tuple values to variables in the expression. If the expression returns `true`, the Tuple is removed from the stream.

```
// incoming -> "ip", "time", "method", "event", "status", "size"

ExpressionFilter filter =
  new ExpressionFilter( "status != 200", Integer.TYPE );

assembly = new Each( assembly, new Fields( "status" ), filter );

// outgoing -> "ip", "time", "method", "event", "status", "size"
```

In this example, every line in the Apache log that does not have a status of "200" is filtered out. ExpressionFilter coerces the value into the specified type if necessary to make the comparison - in this case, coercing the status String into an `int`.

As of Cascading 2.2, along with `cascading.operation.expression.ExpressionFilter` and `cascading.operation.expression.ExpressionFunction`, two new operations have been added to support multi-line Java code, `cascading.operation.expression.ScriptFilter` and `cascading.operation.expression.ScriptFunction`. See the relevant Javadoc for details on usage.

9.8 XML Operations

To use XML Operations in a Cascading application, include the `cascading-xml-x.y.z.jar` in the project. When using the `TagSoupParser` operation, this module requires the TagSoup library, which provides support for HTML and XML "tidying". More information is available at the TagSoup website, http://home.ccil.org/~cowan/XML/tagsoup/.

XPathParser
: The `cascading.operation.xml.XPathParser` function uses one or more XPath expressions, passed into the constructor, to extract one or more node values from an XML document contained in the passed Tuple argument, and places the result(s) into one or more new fields in the current Tuple. In this way, it effectively parses an XML document into a table of fields, creating one Tuple field value for every given XPath expression. The `Node` is converted to a String type containing an XML document. If only the text values are required, search on the `text()` nodes, or consider using XPathGenerator to handle multiple `NodeList` values. If the returned result of an XPath expression is a `NodeList`, only the first `Node` is used for the field value and the rest are ignored.

XPathGenerator
: Similar to XPathParser, the `cascading.operation.xml.XPathGenerator` function emits a new `Tuple` for every `Node` returned by the given XPath expression from the XML in the current Tuple.

XPathFilter
: The filter `cascading.operation.xml.XPathFilter` removes a Tuple if the specified XPath expression returns `false`. Set the removeMatch parameter to `true` if the filter should be reversed, i.e., to keep only those Tuples where the XPath expression returns `true`.

TagSoupParser
: The `cascading.operation.xml.TagSoupParser` function uses the TagSoup library to convert incoming HTML to clean XHTML. Use the `setFeature( feature, value )` method to set TagSoup-specific features, which are documented on the TagSoup website.

9.9 Assertions

Cascading Stream Assertions are used to build robust reusable pipe assemblies. If desired, they can be planned out of a Flow instance at runtime. For more information, see the section on Stream Assertions. Below we describe the Assertions available in the core library.

AssertEquals

The `cascading.operation.assertion.AssertEquals` Assertion asserts that the number of values given on the constructor is equal to the number of argument Tuple values, and that each constructor value `.equals()` its corresponding argument value.

AssertNotEquals

The `cascading.operation.assertion.AssertNotEquals` Assertion asserts that the number of values given on the constructor is equal to the number of argument Tuple values and that each constructor value is not `.equals()` to its corresponding argument value.

AssertEqualsAll

The `cascading.operation.assertion.AssertEqualsAll` Assertion asserts that every value in the argument Tuple `.equals()` the single value given on the constructor.

AssertExpression

The `cascading.operation.assertion.AssertExpression` Assertion dynamically resolves a given Java expression (see Expression Operations) using argument Tuple values. Any Tuple that returns `true` for the given expression passes the assertion.

AssertMatches

The `cascading.operation.assertion.AssertMatches` Assertion matches the given regular expression pattern String against the entire argument Tuple. The comparison is made possible by concatenating all the fields of the Tuple, separated by the TAB character (\t). If a match is found, the Tuple passes the assertion.

AssertMatchesAll

The `cascading.operation.assertion.AssertMatchesAll` Assertion matches the given regular expression pattern String against each argument Tuple value individually.

AssertNotNull

The `cascading.operation.assertion.AssertNotNull` Assertion asserts that every position/field in the argument Tuple is not `null`.

AssertNull

The `cascading.operation.assertion.AssertNull` Assertion asserts that every position/field in the argument Tuple is `null`.

AssertSizeEquals

The `cascading.operation.assertion.AssertSizeEquals` Assertion asserts that the current Tuple in the tuple stream is exactly the given size. Size, here, is the number of fields in the Tuple, as returned by `Tuple.size()`. Note that some or all fields may be `null`.

AssertSizeLessThan

The `cascading.operation.assertion.AssertSizeLessThan` Assertion asserts that the current Tuple in the stream has a size less than (<) the given size. Size, here, is the number of fields in the Tuple, as returned by `Tuple.size()`. Note that some or all fields may be `null`.

AssertSizeMoreThan

The `cascading.operation.assertion.AssertSizeMoreThan` Assertion asserts that the current Tuple in the stream has a size greater than (>) the given size. Size, here, is the number of fields in the Tuple, as returned by `Tuple.size()`. Note that some or all fields may be `null`.

AssertGroupSizeEquals

The `cascading.operation.assertion.AssertGroupSizeEquals` Group Assertion asserts that the number of items in the current grouping is equal to (==) the given size. If a pattern String is given, only grouping keys that match the regular expression will have this assertion applied where multiple key values are delimited by a TAB character.

AssertGroupSizeLessThan

The `cascading.operation.assertion.AssertGroupSizeEquals` Group Assertion asserts that the number of items in the current grouping is less than (<) the given size. If a pattern String is given, only grouping keys that match the regular expression will have this assertion applied where multiple key values are delimited by a TAB character.

AssertGroupSizeMoreThan

The `cascading.operation.assertion.AssertGroupSizeEquals` Group Assertion asserts that the number of items in the current grouping is greater than (>) the given size. If a pattern String is given, only grouping keys that match the regular expression will have this assertion applied where multiple key values are delimited by a TAB character.

9.10 Logical Filter Operators

The logical `Filter` operators allow you to combine multiple filters to run in a single Pipe, instead of chaining multiple Pipes together to get the same logical result.

And

The `cascading.operation.filter.And Filter` performs a logical "and" on the results of the constructor-provided `Filter` instances. That is, if `Filter#isRemove()` returns `true` for all of the given instances, this filter returns `true`.

Or

The `cascading.operation.filter.Or Filter` performs a logical "or" on the results of the constructor-provided `Filter` instances. That is, if `Filter#isRemove()` returns `true` for any of the given instances, this filter returns `true`.

Not

The `cascading.operation.filter.Not Filter` performs a logical "not" (negation) on the results of the constructor-provided `Filter` instance. That is, if `Filter#isRemove()` returns `true` for the given instance, this filter returns `false`, and if `Filter#isRemove()` returns `false` for the given instance, this filter returns `true`.

Xor

The `cascading.operation.filter.Xor Filter` performs a logical "xor" (exclusive or) on the results of the constructor-provided `Filter` instances. Xor can only be applied to two instances at a time. It returns `true` if the two instances have different truth values, and `false` if they have the same truth value. That is, if `Filter.isRemove()` returns `true` for both, or returns `false` for both, this filter returns `false`; otherwise it returns `true`.

```
// incoming -> "ip", "time", "method", "event", "status", "size"

FilterNull filterNull = new FilterNull();
RegexFilter regexFilter = new RegexFilter( "(GET|HEAD|POST)" );

And andFilter = new And( filterNull, regexFilter );

assembly = new Each( assembly, new Fields( "method" ), andFilter );

// outgoing -> "ip", "time", "method", "event", "status", "size"
```

Example 9.1 Combining Filters

The example above performs a logical "and" on the two filters. Both must be satisfied for the data to pass through this one Pipe.

9.11 Buffers

As of Cascading 2.2, the FirstNBuffer Buffer is provided as an optimized means to determine the top N elements in a grouping.

FirstNBuffer

The `cascading.operation.buffer.FirstNBuffer Buffer` returns the first N tuples seen in a given grouping. Unlike the `cascading.pipe.assembly.FirstBy AggregateBy` and `cascading.operation.aggregator.First Aggregator`, `FirstNBuffer` will stop iterating the available tuples when the top N condition is met. `FirstNBuffer` is used by `cascading.pipe.assembly.Unique`.

10. Built-in Assemblies

There are a number of helper SubAssemblies provided by the core cascading library.

As of Cascading 2.2, many of the below assemblies can optionally ignore null values. This allows for an optional but closer resemblance to how similar functions in SQL perform.

10.1 AggregateBy

The `cascading.pipe.assembly.AggregateBy` SubAssembly is an implementation of the Partial Aggregation pattern, and is the base class for built-in and custom partial aggregation implementations like `AverageBy` or `CountBy`.

Generally the AggregateBy class is used to combine multiple AggregateBy subclasses into a single Pipe.

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );

// note we do not pass the parent assembly Pipe in
Fields valueField = new Fields( "size" );
Fields sumField = new Fields( "total-size", long.class );
SumBy sumBy = new SumBy( valueField, sumField );

Fields countField = new Fields( "num-events" );
CountBy countBy = new CountBy( countField );

assembly = new AggregateBy( assembly, groupingFields, sumBy, countBy );
```

Example 10.1 Composing partials with AggregateBy

To create a custom partial aggregation, subclass the AggregateBy class and implement the appropriate internal interfaces. See the Javadoc for details.

AverageBy

The `cascading.pipe.assembly.AverageBy` SubAssembly performs an average over the given `valueFields` and returns the result in the `averageField` field. AverageBy may be combined with other AggregateBy subclasses so they may be executed simultaneously over the same grouping.

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );
Fields valueField = new Fields( "size" );
Fields avgField = new Fields( "avg-size" );
assembly = new AverageBy( assembly, groupingFields, valueField, avgField );
```

Example 10.2 Using AverageBy

CountBy

The `cascading.pipe.assembly.CountBy` SubAssembly performs a count over the given `groupingFields` and returns the result in the `countField` field. CountBy may be combined with other AggregateBy subclasses so they may be executed simultaneously over the same grouping.

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );
Fields countField = new Fields( "count" );
assembly = new CountBy( assembly, groupingFields, countField );
```

Example 10.3 Using CountBy

SumBy

The `cascading.pipe.assembly.SumBy` SubAssembly performs a sum over the given `valueFields` and returns the result in the `sumField` field. SumBy may be combined with other AggregateBy subclasses so they may be executed simultaneously over the same grouping.

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );
Fields valueField = new Fields( "size" );
Fields sumField = new Fields( "total-size" );
assembly =
  new SumBy( assembly, groupingFields, valueField, sumField, long.class );
```

Example 10.4 Using SumBy

FirstBy

The `cascading.pipe.assembly.FirstBy` SubAssembly is used to return the first encountered value in the given `valueFields`. FirstBy may be combined with other AggregateBy subclasses so they may be executed simultaneously over the same grouping.

```
Pipe assembly = new Pipe( "assembly" );

// ...
Fields groupingFields = new Fields( "date" );
Fields valueField = new Fields( "size" );

// we want the largest size in this grouping
valueField.setComparator( "size", new LongComparator() );

assembly =
  new FirstBy( assembly, groupingFields, valueField );
```

Example 10.5 Using FirstBy

Note if the `valueFields` Fields instance has field comparators, they will be used to sort the argument values to influence what values are seen first. Otherwise the fields will not be sorted in any deterministic order.

10.2 Coerce

The `cascading.pipe.assembly.SumBy` SubAssembly is used to coerce a set of values from one type to another type - for example, to convert the field `age` from a `String` to an `Integer`.

```
// incoming -> first, last, age

assembly =
  new Coerce( assembly, new Fields( "age" ), Integer.class );

// outgoing -> first, last, age
```

Example 10.6 Using Coerce

10.3 Discard

The `cascading.pipe.assembly.Discard` SubAssembly is used to shape the Tuple stream by discarding all fields given on the constructor. All fields not listed are retained.

```
// incoming -> first, last, age

assembly = new Discard( assembly, new Fields( "age" ) );

// outgoing -> first, last
```

Example 10.7 Using Discard

10.4 Rename

The `cascading.pipe.assembly.Rename` SubAssembly is used to rename a field.

```
// incoming -> first, last, age

assembly =
  new Rename( assembly, new Fields( "age" ), new Fields( "years" ) );

// outgoing -> first, last, years
```

Example 10.8 Using Rename

10.5 Retain

The `cascading.pipe.assembly.Retain` SubAssembly is used to shape the Tuple stream by retaining all fields given on the constructor. All fields not listed are discarded.

```
// incoming -> first, last, age

assembly = new Retain( assembly, new Fields( "first", "last" ) );

// outgoing -> first, last
```

Example 10.9 Using Retain

10.6 Unique

The `cascading.pipe.assembly.Unique` SubAssembly is used to remove duplicate values in a Tuple stream. Uniqueness is determined by the values of all fields listed in `uniqueFields`. Thus to find all distinct Tuples in a Tuple stream, use `Fields.ALL` as the `uniqueFields` argument.

```
// incoming -> first, last

assembly = new Unique( assembly, new Fields( "first", "last" ) );

// outgoing -> first, last
```

Example 10.10 Using Unique

As of Cascading 2.2, `Unique` uses the `FirstNBuffer` to more efficiently determine unique values.

11. Best Practices

11.1 Unit Testing

Discrete testing of all Operations, pipe assemblies, and applications is a must. The `cascading.CascadingTestCase` provides a number of static helper methods.

When testing custom Operations, use the `invokeFunction()`, `invokeFilter()`, `invokeAggregator()`, and `invokeBuffer()` methods.

When testing Flows, use the `validateLength()` methods. There are quite a few of them, and collectively they offer great flexibility. All of them read the sink tap, validate that it is the correct length and has the correct Tuple size, and check to see whether the values match a given regular expression pattern.

As of Cascading 2, it is possible to write tests that are independent of the underlying platform. Any unit test should subclass `cascading.PlatformTestCase` located in the `cascading-platform-x.y.z-tests.jar` jar file. Any platform to be tested against should be added to the `CLASSPATH` as well. `PlatformTestCase` will search the `CLASSPATH` for all available platforms and run each test on the subclass against each platform found.

See the Cascading platform unit tests for examples.

For Maven users, be sure to add the `tests` classifier to any dependencies. Note that the `cascading-platform` project has no main code, but does have only tests, so it must be retrieved via the `tests` classifier.

11.2 Flow Granularity

Although using one large `Flow` may result in slightly more efficient performance, it's advisable to use a more modular and flexible approach, creating medium or small Flows with well-defined responsibilities, and passing all the resulting interdependent Flows to a `Cascade` to sequence and execute as a single unit. Similarly, using the `TextDelimited` `Scheme` (or any custom format for long-term archival) between `Flow` instances allows you to hand off intermediate data to other systems for reporting or QA purposes, incurring a minimal performance penalty while remaining compatible with other tools.

11.3 SubAssemblies, not Factories

When developing your applications, use `SubAssembly` subclasses, not "factory" methods. The resulting code is much easier to read and test.

It's worth noting that the `Object` constructors are "factories", so there isn't much reason to build frameworks to duplicate what a constructor already does. Of course there are exceptional cases in which you don't have the option to use a `SubAssembly`, but in practice they are rare.

11.4 Logical Responsibilities for SubAssemblies

SubAssembies provide a very convenient means to co-locate similar or related responsibilities into a single place. For example, it's simple to use a `ParsingSubAssembly` and a `RulesSubAssembly`, where the first is responsible solely for parsing incoming `Tuple` streams (log files for example), and the second applies rules to decide whether a given `Tuple` should be discarded or marked as bad.

Additionally, in your unit tests you can create a `TestAssertionsSubAssembly` that simply inlines various `ValueAssertions` and `GroupAssertions`. The practice of inlining Assertions directly in your SubAssemblies is also important, but sometimes it makes sense to have more tests outside of the business logic.

11.5 Java Operators in Field Names

There are a few Operations in Cascading (e.g., `ExpressionFunction` and `ExpressionFilter`) that compile and apply Java expressions on the fly. In these expressions, Operation argument field names are used as variable names in the expression. For this reason, take care to create field names that don't contain characters which will cause compilation errors if they are used in an expression. For example, "first-name" is a valid field name for use with Cascading, but might result in the expression `first-name.trim()`, which will cause a compilation error.

11.6 Debugging Planner Failures

The `FlowConnector` will sometimes fail when attempting to plan a `Flow`. If the error message given by `PlannerException` is vague, use the method `PlannerException.writeDOT()` to export a representation of the internal pipe assembly. DOT files can be opened by GraphViz and OmniGraffle. These plans are only partial, but you will be able to see where the Cascading planner failed.

Note that you can also create a DOT file from a `Flow`, by using `Flow.writeDOT()`.

See Debugging Hadoop.

11.7 Optimizing Joins

When joining two streams via a `CoGroup` `Pipe`, try to put the largest of the streams in the leftmost argument to the `CoGroup`. The reason for this is that joining multiple streams requires some accumulation of values before the join operator can begin, but the leftmost stream is not accumulated, so this technique should improve the performance of most joins.

11.8 Debugging Streams

When creating complex assemblies, it's safe to embed `Debug` operations (see Debug Function) at appropriate debug levels as needed. Use the planner to remove them at runtime for production and staging runs, to avoid wasting resources.

See Debugging Hadoop.

11.9 Handling Good and Bad Data

It's very common when processing raw data streams to encounter data that is corrupt or malformed in some way. For instance, bad content may be fetched from the web via a crawler upstream, or a bug may have leaked into a browser widget somewhere that sends user behavior information back for analysis. Whatever the cause, it's a good practice to define a set of rules for identifying and discarding questionable records.

It is tempting to simply throw an exception and have a Trap capture the offending `Tuple`, but Traps were not designed as a filtering mechanism, and consequently much valuable information would be lost.

Instead of traps, use filters. Create a `SubAssembly` that applies rules to the stream by setting a binary field that marks the tuple as good or bad. After all the rules are applied, split the stream based on the value of the good or bad `Boolean` value. Consider setting a reason field that states why the Tuple was marked bad.

11.10 Maintaining State in Operations

When creating custom Operations (`Function`, `Filter`, `Aggregator`, or `Buffer`) do not store operation state in class fields.

For example, if implementing a custom "counter" `Aggregator`, do not create a field named "count" and increment it on every `Aggregator.aggregate()` call. There is no guarantee that your Operation will be called from a single thread in a JVM - and future versions of Hadoop or Cascading local mode might execute the same operation from multiple threads.

11.11 Custom Types

Passing a custom class through a Tuple stream is generally frowned upon. It leads to the coupling of custom Operations to particular types, and it removes opportunities for reducing the amount of data that passes over the network.

The first objection can be overcome with a little work. When using a custom type that has multiple instance fields, try to provide Functions that can promote a value from the custom object to a position in a Tuple, or demote the Tuple value for a particular field back into the custom type. This lets you use existing operations like `ExpressionFunction` or `RegexFilter` to operate on values owned by a custom type.

For example, if you have a `Person` object, create a Function named `GetPersonAge` that takes Person as an argument and returns just the age. The next operation can then Filter all Persons based on their age. This may seem more difficult and less effiicient, but it keeps your application flexible and avoids duplicating existing operations. (The only alternative here is to create a `PersonAgeFilter`, which becomes one more thing to test.)

11.12 Fields Constants

Instead of having String field names strewn about, create an Interface that holds a constant value for each field name:

```
public static Fields FIRST_NAME = new Fields( "firstname" );
```

Using the Fields class, instead of String, allows for building more complex constants:

```
public static Fields NAME = FIRST_NAME.append( LAST_NAME );
```

11.13 Checking the Source Code

When in doubt, look at the Cascading source code. If something is not documented in this User Guide or Javadoc, and it's a feature of Cascading, the feature source code or unit tests will give you clear instructions on what to do or expect.

12. Extending Cascading

12.1 Scripting

The Cascading API was designed with scripting in mind. Any Java-compatible scripting language can import and instantiate Cascading classes, create pipe assemblies and flows, and execute those flows. And if the scripting language in question supports Domain Specific Language (DSL) creation, users can create their own DSLs to handle common idioms.

The Cascading website includes information on scripting language bindings that are publicly available.

12.2 Custom Types and Serialization

The `Tuple` class is a generic container for all `java.lang.Object` instances. Thus any primitive value or custom Class can be stored in a `Tuple` instance - that is, returned by a `Function`, `Aggregator`, or `Buffer` as a result value.

But for this to work when using the Cascading Hadoop mode, any Class that isn't a primitive type or a Hadoop `Writable` type requires a corresponding Hadoop serialization class registered in the Hadoop configuration files for your cluster. Hadoop `Writable` types work because there is already a generic serialization implementation built into Hadoop. See the Hadoop documentation for information on registering a new serialization helper or creating `Writable` types. Registered serialization implementations are automatically inherited by Cascading.

During serialization and deserialization of `Tuple` instances that contain custom types, the Cascading `Tuple` serialization framework must store the class name (as a `String`) before serializing the custom object. This can be very space-inefficient. To overcome this, custom types can add the `SerializationToken` Java annotation to the custom type class. The `SerializationToken` annotation expects two arrays - one of integers that are used as tokens, and one of Class name strings. Both arrays must be the same size. The integer tokens must all have values of 128 or greater, since the first 128 values are reserved for internal use.

During serialization and deserialization, the token values are used instead of the `String` Class names, in order to reduce the amount of storage used.

Serialization tokens may also be stored in the Hadoop config files or set as a property passed to the `FlowConnector`, with the property name `cascading.serialization.tokens`. The value of this property is a comma separated list of `token=classname` values.

Note that Cascading natively serializes/deserializes all primitives and byte arrays (`byte[]`), if the developer registers the `BytesSerialization` class by using `TupleSerializationProps.addSerialization(properties, BytesSerialization.class.getName()`. The token 127 is used for the Hadoop `BytesWritable` class.

By default, Cascading uses lazy deserialization on Tuple elements during comparisons when Hadoop sorts keys during the "shuffle" phase.

Cascading supports custom serialization for custom types, as well as lazy deserialization of custom types during comparisons. This is accomplished by implementing the `StreamComparator` interface. See the Javadoc for detailed instructions on implemention, and the unit tests for examples.

12.3 Custom Comparators and Hashing

Frequently, objects in one `Tuple` are compared to objects in a second `Tuple`. This is especially true during the sort phase of `GroupBy` and `CoGroup` in Cascading Hadoop mode . By default, Hadoop and Cascading use the native `Object` methods `equals()` and `hashCode()` to compare two values and get a consistent hash code for a given value, respectively.

To override this default behavior, you can create a custom `java.util.Comparator` class to perform comparisons on a given field in a Tuple. For instance, to secondary-sort a collection of custom `Person` objects in a `GroupBy`, use the `Fields.setComparator()` method to designate the custom `Comparator` to the `Fields` instance that specifies the sort fields.

Alternatively, you can set a default `Comparator` to be used by a `Flow`, or used locally on a given `Pipe` instance. There are two ways to do this. Call `FlowProps.setDefaultTupleElementComparator()` on a `Properties` instance, or use the property key `cascading.flow.tuple.element.comparator`.

If the hash code must also be customized, the custom Comparator can implement the interface `cascading.tuple.Hasher`. For more information, see the Javadoc.

13. Cookbook

This chapter demonstrates some common idioms used in Cascading applications.

13.1 Tuples and Fields

Copy a Tuple instance

```
Tuple original = new Tuple( "john", "doe" );

// call copy constructor
Tuple copy = new Tuple( original );

assert copy.getObject( 0 ).equals( "john" );
assert copy.getObject( 1 ).equals( "doe" );
```

Nest a Tuple instance within a Tuple

```
Tuple parent = new Tuple();
parent.add( new Tuple( "john", "doe" ) );

assert ( (Tuple) parent.getObject( 0 ) ).getObject( 0 ).equals( "john" );
assert ( (Tuple) parent.getObject( 0 ) ).getObject( 1 ).equals( "doe" );
```

Build a longer Fields instance

```
Fields first = new Fields( "first" );
Fields middle = new Fields( "middle" );
Fields last = new Fields( "last" );

Fields full = first.append( middle ).append( last );
```

Remove a field from a longer Fields instance

```
Fields full = new Fields( "first", "middle", "last" );

Fields firstLast = full.subtract( new Fields( "middle" ) );
```

13.2 Stream Shaping

Split (branch) a Tuple Stream

```
Pipe pipe = new Pipe( "head" );
```

```
pipe = new Each( pipe, new SomeFunction() );
// ...

// split left with the branch name 'lhs'
Pipe lhs = new Pipe( "lhs", pipe );
lhs = new Each( lhs, new SomeFunction() );
// ...

// split right with the branch name 'rhs'
Pipe rhs = new Pipe( "rhs", pipe );
rhs = new Each( rhs, new SomeFunction() );
// ...
```

Copy a field value

```
Fields argument = new Fields( "field" );
Identity identity = new Identity( new Fields( "copy" ) );

// identity copies the incoming argument to the result tuple
pipe = new Each( pipe, argument, identity, Fields.ALL );
```

Discard (drop) a field

```
// incoming -> "keepField", "dropField"
pipe = new Discard( pipe, new Fields( "dropField" ) );
// outgoing -> "keepField"
```

Retain (keep) a field

```
// incoming -> "keepField", "dropField"
pipe = new Retain( pipe, new Fields( "keepField" ) );
// outgoing -> "keepField"
```

Rename a field

```
// a simple SubAssembly that uses Identity internally
pipe = new Rename( pipe, new Fields( "from" ), new Fields( "to" ) );
```

Coerce field values from Strings to primitives

```
Fields fields = new Fields( "longField", "booleanField" );
Class types[] = new Class[]{long.class, boolean.class};

// convert to given type
pipe = new Coerce( pipe, fields, types );
```

Insert constant values into a stream

```
Fields fields = new Fields( "constant1", "constant2" );
Insert function = new Insert( fields, "value1", "value2" );

pipe = new Each( pipe, function, Fields.ALL );
```

13.3 Common Operations

Parse a String date/time value

```
// convert string date/time field to a long
// milliseconds "timestamp" value
String format = "yyyy:MM:dd:HH:mm:ss.SSS";
DateParser parser = new DateParser( new Fields( "ts" ), format );

pipe = new Each( pipe, new Fields( "datetime" ), parser, Fields.ALL );
```

Format a time-stamp to a date/time value

```
// convert a long milliseconds "timestamp" value to a string
String format = "HH:mm:ss.SSS";
DateFormatter formatter =
  new DateFormatter( new Fields( "datetime" ), format );

pipe = new Each( pipe, new Fields( "ts" ), formatter, Fields.ALL );
```

13.4 Stream Ordering

Remove duplicate tuples in a stream

```
// remove all duplicates from the stream
pipe = new Unique( pipe, Fields.ALL );
```

Create a list of unique values

```
// narrow stream to just ips
pipe = new Retain( pipe, new Fields( "ip" ) );
// find all unique 'ip' values
pipe = new Unique( pipe, new Fields( "ip" ) );
```

Find first occurrence in time of a unique value

```
// group on all unique 'ip' values
// secondary sort on 'datetime', natural order is in ascending order
```

```
pipe = new GroupBy( pipe, new Fields( "ip" ), new Fields( "datetime" ) );
// take the first 'ip' tuple in the group which has the
// oldest 'datetime' value
pipe = new Every( pipe, Fields.ALL, new First(), Fields.RESULTS );
```

13.5 API Usage

Pass properties to a custom Operation

```
// set property on Flow
Properties properties = new Properties();
properties.put( "key", "value" );
FlowConnector flowConnector = new HadoopFlowConnector( properties );
// ...

// get the property from within an Operation (Function, Filter, etc)
String value = (String) flowProcess.getProperty( "key" );
```

Bind multiple sources and sinks to a Flow

```
Pipe headLeft = new Pipe( "headLeft" );
// do something interesting

Pipe headRight = new Pipe( "headRight" );
// do something interesting

// merge the two input streams
Pipe merged = new GroupBy( headLeft, headRight, new Fields( "common" ) );
// ...

// branch the merged stream
Pipe tailLeft = new Pipe( "tailLeft", merged );
// filter out values to the left
tailLeft = new Each( tailLeft, new SomeFilter() );

Pipe tailRight = new Pipe( "tailRight", merged );
// filter out values to the right
tailRight = new Each( tailRight, new SomeFilter() );

// source taps
Scheme inLeftScheme =
  new TextDelimited( new Fields( "some-fields" ) );
Tap sourceLeft = new Hfs( inLeftScheme, "some/path" );

Scheme inRightScheme =
```

```
  new TextDelimited( new Fields( "some-fields" ) );
Tap sourceRight = new Hfs( inRightScheme, "some/path" );

// sink taps
Scheme outLeftScheme =
  new TextDelimited( new Fields( "some-fields" ) );
Tap sinkLeft = new Hfs( outLeftScheme, "some/path" );

Scheme outRightScheme =
  new TextDelimited( new Fields( "some-fields" ) );
Tap sinkRight = new Hfs( outRightScheme, "some/path" );

FlowDef flowDef = new FlowDef()
  .setName( "flow-name" );

// bind source Taps to Pipe heads
flowDef
  .addSource( headLeft, sourceLeft )
  .addSource( headRight, sourceRight );

// bind sink Taps to Pipe tails
flowDef
  .addSink( tailLeft, sinkLeft )
  .addTailSink( tailRight, sinkRight );

// ALTERNATIVELY ...

// add named source Taps
// the head pipe name to bind to
flowDef
  .addSource( "headLeft", sourceLeft )    // headLeft.getName()
  .addSource( "headRight", sourceRight ); // headRight.getName()

// add named sink Taps
flowDef
  .addSink( "tailLeft", sinkLeft )    // tailLeft.getName()
  .addSink( "tailRight", sinkRight ); // tailRight.getName()

// add tails -- heads are reachable from the tails
flowDef
  .addTail( tailLeft )
  .addTail( tailRight );

// set property on Flow
FlowConnector flowConnector = new HadoopFlowConnector();
```

```
Flow flow = flowConnector.connect( flowDef );
```

14. How It Works

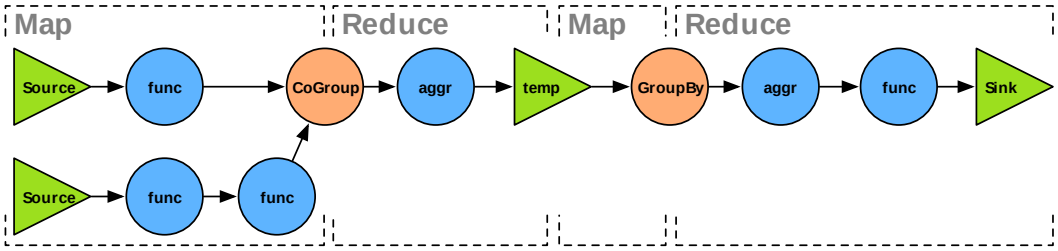
14.1 MapReduce Job Planner

The Hadoop MapReduce Job Planner is an internal feature of Cascading.

When a collection of functions, splits, and joins are all tied up together into a "pipe assembly", the FlowConnector object is used to create a new Flow instance against input and output data paths. This Flow is a single Cascading job.

Internally, the FlowConnector employs an intelligent planner to convert the pipe assembly to a graph of dependent MapReduce jobs that can be executed on a Hadoop cluster.

All this happens behind the scenes - as does the scheduling of the individual MapReduce jobs, and the cleanup of intermediate data sets that bind the jobs together.



The diagram above shows how a typical Flow is partitioned into MapReduce jobs. Every job is delimited by a temporary file that serves as the sink from the first job and the source for the next.

To create a visualization of how your Flows are partitioned, call the `Flow#writeDOT()` method. This writes a DOT [http://en.wikipedia.org/wiki/DOT_language] file out to the path specified, which can be viewed in a graphics package like OmniGraffle or Graphviz.

14.2 The Cascade Topological Scheduler

Cascading has a simple class, `Cascade` , that executes a collection of Cascading Flows on a target cluster in dependency order.

Consider the following example.

- Flow 1 reads input file A and outputs B.
- Flow 2 expects input B and outputs C and D.
- Flow 3 expects input C and outputs E.

A `Cascade` is constructed through the `CascadeConnector` class, by building an internal graph that makes each Flow a "vertex", and each file an "edge". A topological walk on this graph will touch each vertex in order of its dependencies. When a vertex has all its incoming edges (i.e., files) available, it is scheduled on the cluster.

In the example above, Flow 1 goes first, Flow 2 goes second, and Flow 3 is last.

If two or more Flows are independent of one another, they are scheduled concurrently.

And by default, if any outputs from a Flow are newer than the inputs, the Flow is skipped. The assumption is that the Flow was executed recently, since the output isn't stale. So there is no reason to re-execute it and use up resources or add time to the job. This is similar behavior a compiler would exhibit if a source file wasn't updated before a recompile.

This is very handy if you have a large set of jobs, with varying interdependencies between them, that needs to be executed as a logical unit. Just pass them to the CascadeConnector and let it sort them all out.