Cascading - User Guide

Concurrent, Inc

V 1.2
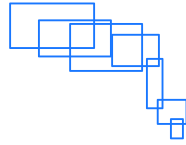
Published October, 2010

Table of Contents

1. Cascading

1.1 What is Cascading?

Cascading is a Query API and Query Planner used for defining, sharing, and executing data processing workflows on a distributed data grid or cluster.

Cascading relies on Apache Hadoop. To use Cascading, Hadoop must be installed locally for development and testing, and a Hadoop cluster must be deployed for production applications.

Cascading greatly simplifies the complexities with Hadoop application development, job creation, and job scheduling.

1.2 Who should use Cascading?

Cascading was developed to allow organizations to rapidly develop complex data processing applications. These applications come in two extremes.

On one hand, there is too much data for a single computing system to manage effectively. Developers have decided to adopt Apache Hadoop as the base computing infrastructure, but realize that developing reasonably useful applications on Hadoop is not trivial. Cascading eases the burden on developers by allowing them to rapidly create, refactor, test, and execute complex applications that scale linearly across a cluster of computers.

On the other hand, managers and developers realize the complexity of the processes in their data center is getting out of hand with one-off data-processing applications living wherever there is enough disk space or available CPU. Subsequently they have decided to adopt Apache Hadoop to gain access to its "Global Namespace" file system which allows for a single reliable storage framework. Cascading eases the learning curve for developers to convert their existing applications for execution on a Hadoop cluster. It further allows for developers to create reusable libraries and application for use by analysts who need to extract data from the Hadoop file system.

Cascading was designed to support three user roles. The application Executor, process Assembler, and the operation Developer.

The application Executor is someone, a developer or analyst, or some system (like a cron job) which runs a data processing application on a given cluster. This is typically done via the command line using a pre-packaged Java Jar file compiled against the Apache Hadoop and Cascading libraries. This application may accept command line parameters to customize it for an given execution and generally results in a set of data the user will export from the Hadoop file system for some specific purpose.

The process Assembler is someone who assembles data processing workflows into unique applications. This is generally a development task of chaining together operations that act on input data sets to produce one or more output data sets. This task can be done using the raw Java Cascading API or via a scripting language like Cascalog/Clojure, Groovy, JRuby, or Jython.

The operation Developer is someone who writes individual functions or operations, typically in Java, or reusable sub-assemblies that act on the data that pass 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, simply parsing a string, or performing some complex routine on the argument data using third-party libraries.

All three roles can be a developer, but the API allows for a clean separation of responsibilities for larger organizations that need 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".

To be a little more specific, Hadoop provides a storage layer that holds vast amounts of data, and an execution layer for running an application in parallel across the cluster against parts of the stored data.

The storage layer, 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" ranges from Gigabytes to Petabytes. But it only supports a single writer. And random access to the data is not really possible in an efficient manner either. But this is why it is so performant and reliable. Reliable in part because this restriction allows for the data to be replicated across the cluster reducing the chance of data loss.

The execution layer relies on a "divide and conquer" strategy called MapReduce. MapReduce is beyond the scope of this document, but suffice it to say, it can be so difficult to develop "real world" applications against that Cascading was created to offset the complexity.

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

2. Diving In

Counting words in a document is the most common example presented to new Hadoop (and MapReduce) developers, it is the Hadoop equivalent to the "Hello World" application.

Word counting is where a document is parsed into individual words, and the frequency of those words are counted.

For example, if we counted the last paragraph "is" would be counted twice, and "document" counted once.

In the code example below, we will use Cascading to read each line of text from a file (our document), parse it into words, then count the number of time the word is encountered.

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

Scheme sinkScheme = new TextLine( 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 = new Properties();
FlowConnector.setApplicationJarClass( properties, Main.class );

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

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

Example 2.1 Word Counting

There are a couple things to take away from this example.

First, the pipe assembly is not coupled to the data (the Tap instances) until the last moment before execution. That is, file paths or references are not embedded in the pipe assembly. The pipe assembly remains independent of *which* data it processes until execution. The only dependency is *what* the data looks like, its "scheme", or the field names that make it up.

That brings up fields. Every input and output file has field names associated with it, and every processing element of the pipe assembly either expects certain fields, or creates new fields. This allows the developer to self document their code, and allows the Cascading planner to "fail fast" during planning if a dependency between elements isn't satisfied (used a missing or wrong field name).

It is also important to point out that pipe assemblies are assembled through constructor chaining. This may seem odd but is done for two reasons. It keeps the code more concise. And it prevents developers from creating "cycles" in the resulting pipe assembly. Pipe assemblies are Directed Acyclic Graphs (or DAGs). The Cascading planner cannot handle processes that feed themselves, that have cycles (not to say there are ways around this that are much safer).

Notice 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. This example does not name the tail assembly, but for complex assemblies, tails must be named for reasons described below.

Heads and tails of pipe assemblies generally need names, this is how sources and sinks are "bound" to them during planning. In our example above, there is only one head and one tail, and subsequently only one source and one sink, respectively. So naming in this case is optional, it's obvious what goes where. Naming is also useful for self documenting pipe assemblies, especially where there are splits, joins, and merges in the assembly.

To paraphrase, our example will:

- read each line of text from a file and give it the field name "line",

- parse each "line" into words by the `RegexGenerator` object which in turn returns each word in the field named "word",

- groups on the field named "word" using the `GroupBy` object,

- then counts the number of elements in each grouping using the `Count()` object and stores this value in the "count" field,

- finally the "word" and "count" fields are written out.

3. Data Processing

3.1 Introduction

The Cascading processing model is based on a "pipes and filters" metaphor. The developer uses the Cascading API to assemble pipelines 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 pipeline a pipe assembly, and a series of Tuples passing through a pipe assembly is called a tuple stream.

Pipe assemblies are assembled independently from what data they will process. Before a pipe assembly can be executed, it must be bound to data sources and data sinks, called Taps. The process of binding pipe assemblies to sources and sinks results in a Flow. Flows can be executed on a data cluster like Hadoop.

Finally, many Flows can be grouped together and executed as a single process. If one Flow depends on the output of another Flow, it will not be executed until all its data dependencies are satisfied. This collection of Flows is called a Cascade.

3.2 Pipe Assemblies

Pipe assemblies define what work should be done against a tuple stream, where during runtime tuple streams are read from Tap sources and are written to Tap sinks. Pipe assemblies may have multiple sources and multiple sinks and they can define splits, merges, and joins to manipulate how the tuple streams interact.
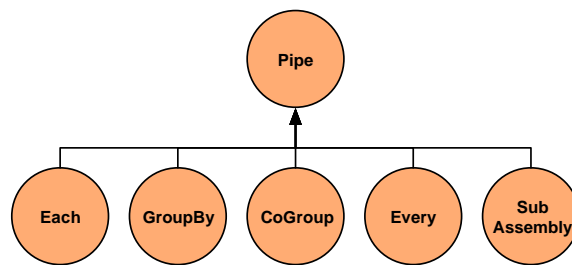
There are only five Pipe types: Pipe, Each, GroupBy, CoGroup, Every, and SubAssembly.

Pipe

>   The `cascading.pipe.Pipe` class is used to name branches of pipe assemblies. These names are used during planning to bind Taps as either sources or sinks (or as traps, an advanced topic). It is also the base class for all other pipes described below.

Each

>   The `cascading.pipe.Each` pipe applies a `Function` or `Filter` Operation to each Tuple that passes through it.

GroupBy

> cascading.pipe.GroupBy manages one input Tuple stream and does exactly as it sounds, that is, groups the stream on selected fields in the tuple stream. GroupBy also allows for "merging" of two or more tuple stream that share the same field names.

CoGroup

> cascading.pipe.CoGroup allows for "joins" on a common set of values, just like a SQL join. The output tuple stream of CoGroup is the joined input tuple streams, where a join can be an Inner, Outer, Left, or Right join.

Every

> The cascading.pipe.Every pipe applies an Aggregator (like count, or sum) or Buffer (a sliding window) Operation to every group of Tuples that pass through it.

SubAssembly

> The cascading.pipe.SubAssembly pipe allows for nesting reusable pipe assemblies into a Pipe class for inclusion in a larger pipe assembly. See the section onSubAssemblies.

Assembling Pipe Assemblies

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

```
// 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() );
```
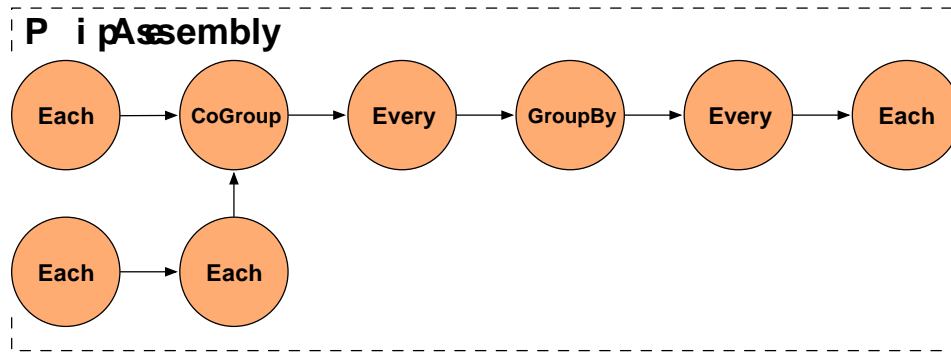
Example 3.1 Chaining Pipes

The above example, if visualized, would look like the diagram below.

Here are some common stream patterns.

Split

A split takes a single stream and sends it down one or more paths. This is simply achieved by passing a given `Pipe` instance to two or more subsequent `Pipe` instances. Note you can use the `Pipe` class and name the branch (branch names are useful for bindingFailure Traps), or with a `Each` class.

Merge

A merge is where two or more streams with the exact same Fields (and types) are treated as a single stream. This is achieved by passing two or more `Pipe` instances to a `GroupBy Pipe` instance.

Join

A join is where two or more streams are connected by one or more common values. See the previous diagram for an example.

Besides defining the paths tuple streams take through splits, merges, grouping, and joining, pipe assemblies also transform and/or filter the stored values in each Tuple. This is accomplished by applying an Operation to each Tuple or group of Tuples as the tuple stream passes through the pipe assembly. To do that, the values in the Tuple typically are given field names, in the same fashion columns are named in a database so that they may be referenced or selected.

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 reused, or developers can create their own custom Operations.

Tuple

In Cascading, we call each record of data a Tuple (`cascading.tuple.Tuple`), and a series of Tuples are a tuple stream. Think of a Tuple as an Array of values where each value can be any `java.lang.Object` Java type (or `byte[]` array). See the section on Custom Types for supporting non-primitive values.

Fields

Fields (`cascading.tuple.Fields`) either declare the field names in a Tuple. Or reference values in a Tuple as a selector. Fields can either be string names ("first_name"), integer positions (`-1` for the last value), or a substitution (`Fields.ALL` to select all values in the Tuple, like an asterisk (`*`) in SQL, seeField Algebra).

Each and Every Pipes

The `Each` and `Every` pipe types are the only pipes that can be used to apply Operations to the tuple stream.

The `Each` pipe applies an Operation to "each" Tuple as it passes through the pipe assembly. The `Every` pipe applies an Operation to "every" group of Tuples as they pass through the pipe assembly, on the tail end of a `GroupBy` or `CoGroup` pipe.

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

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

Both the `Each` and `Every` pipe take a Pipe instance, an argument selector, Operation instance, and a output selector on the constructor. Where each selector is a Fields instance.

The `Each` pipe may only apply `Functions` and `Filters` to the tuple stream as these operations may only operate on one Tuple at a time.

The `Every` pipe may only apply `Aggregators` and `Buffers` to the tuple stream as these operations may only operate on groups of tuples, one grouping at a time.
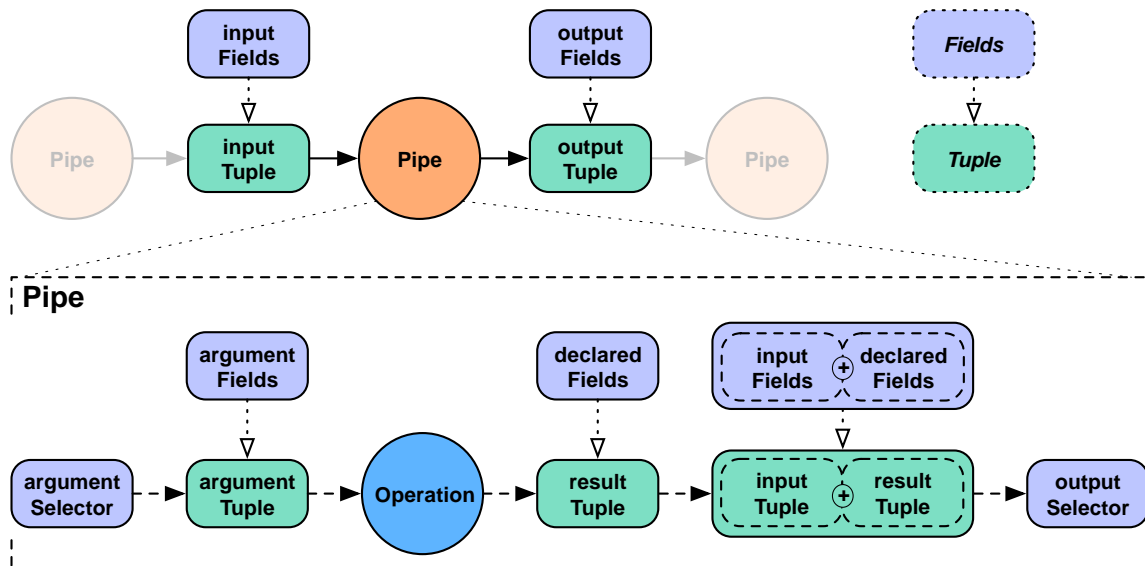
The "argument selector" selects values from the input Tuple to be passed to the Operation as argument values. Most Operations declare result fields, "declared fields" in the diagram. The "output selector" selects the output Tuple from an "appended" version of the input Tuple and the Operation result Tuple. This new output Tuple becomes the input Tuple to the next pipe in the pipe assembly.

Note that if a `Function` or `Aggregator` emits more than one Tuple, this process will be repeated for each result Tuple against the original input Tuple, depending on the output selector, input Tuple values could be duplicated across each output Tuple.
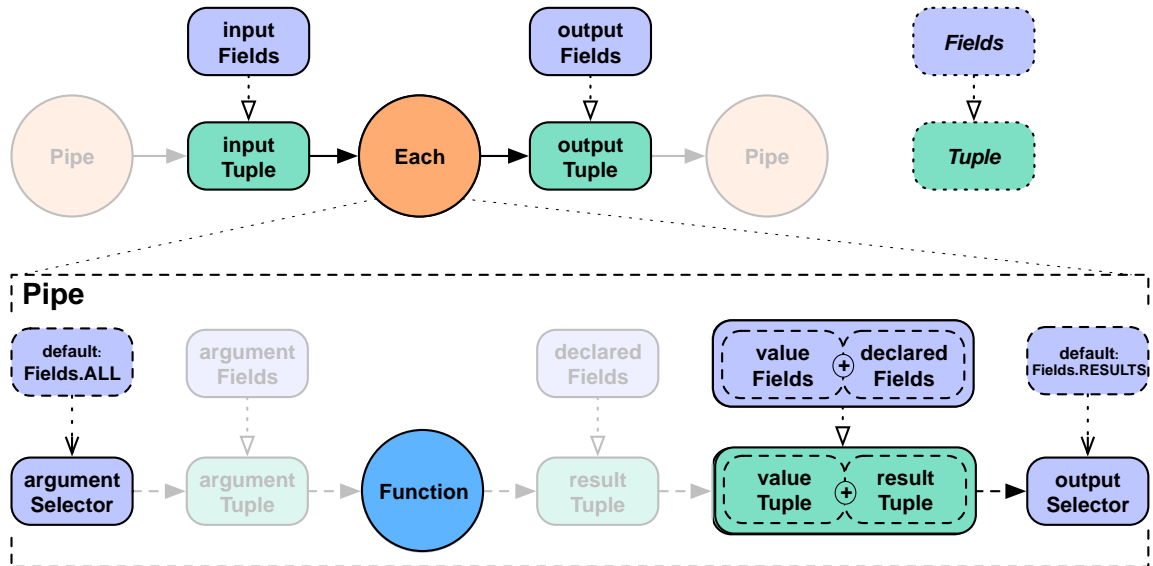
If the argument selector is not given, the whole input Tuple (`Fields.ALL`) is passed to the Operation as argument values. If the result selector is not given on an `Each` pipe, the Operation results are returned by default (`Fields.RESULTS`), replacing the input Tuple values in the tuple stream. This really only applies to `Functions`, as `Filters` either discard the input Tuple, or return the input Tuple intact. There is no opportunity to provide an output selector.
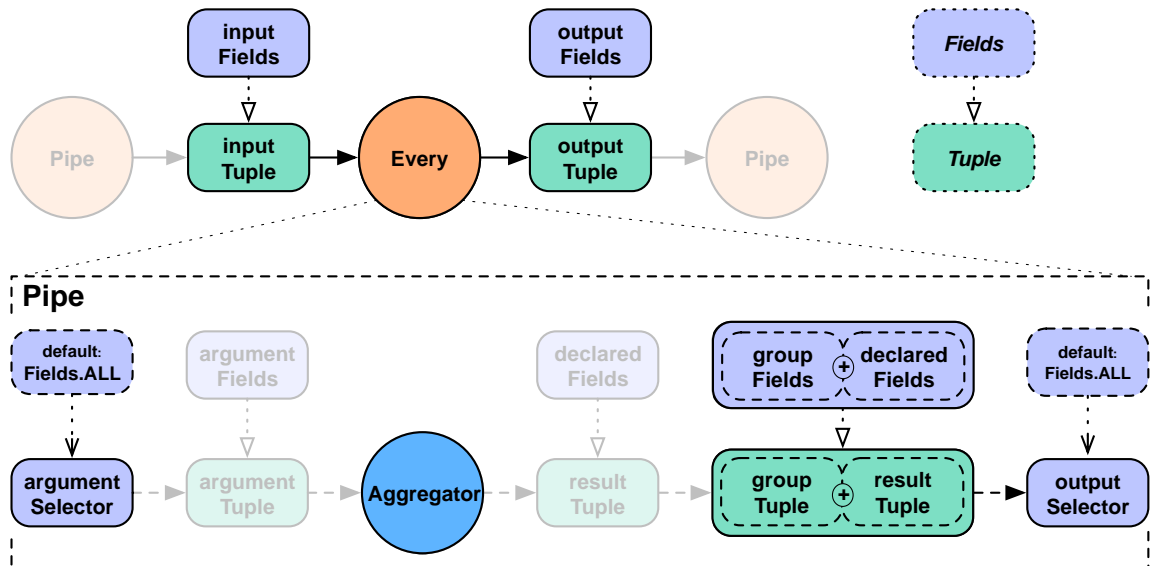
For the `Every` pipe, the Aggregator results are appended to the input Tuple (`Fields.ALL`) by default.

It is important to note that the `Every` pipe associates Aggregator results with the current group Tuple. For example, if you are grouping on the field "department" and counting the number of "names" grouped by that department, the output Fields would be ["department","num_employees"]. This is true for both `Aggregator`, seen above, and `Buffer`.

If you were also adding up the salaries associated with each "name" in each "department", the output Fields would be ["department","num_employees","total_salaries"]. This is only true for chains of `Aggregator` Operations, you may not chain `Buffer` Operations.
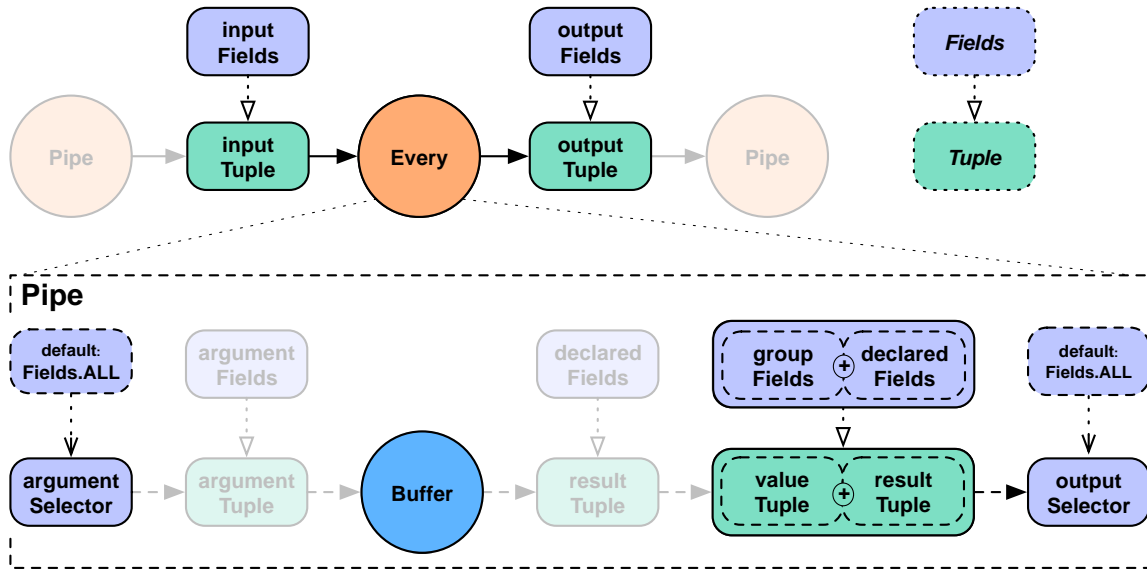
For the `Every` pipe when used with a `Buffer` the behavior is slightly different. Instead of associating the Buffer results with the current grouing Tuple, they are associated with the current values Tuple, just like an `Each` pipe does with a `Function`. This might be slightly more confusing, but provides much more flexibility.

GroupBy and CoGroup Pipes

The `GroupBy` and `CoGroup` pipes serve two roles. First, they emit sorted grouped tuple streams allowing for Operations to be applied to sets of related Tuple instances. Where "sorted" means the tuple groups are emitted from the `GroupBy` and `CoGroup` pipes in sort order of the field values the groups were grouped on.

Second, they allow for two streams to be either merged or joined. Where merging allows for two or more tuple streams originating from different sources to be treated as a single stream. And joining allows two or more streams to be "joined" (in the SQL sense) on a common key or set of Tuple values in a Tuple.

It is not required that an `Every` follow either `GroupBy` or CoGroup, an `Each` may follow immediately after. But an `Every` many not follow an `Each`.

It is important to note, for both `GroupBy` and`CoGroup`, the values being grouped on must be the same type. If your application attempts to `GroupBy` on the field "size", but the value alternates between a `String` and a `Long`, Hadoop will fail internally attempting to apply a Java `Comparator` to them. This also holds true for the secondary sorting sort-by fields in `GroupBy`.

`GroupBy` accepts one or more tuple streams. If two or more, they must all have the same field names (this is also called a merge, see below).

```
Pipe groupBy = new GroupBy( assembly, new Fields( "group1", "group2" ) );
```

Example 3.2 Grouping a Tuple Stream

The example above simply creates a new tuple stream where Tuples with the same values in "group1" and "group2" can be processed as a set by an `Aggregator` or `Buffer` Operation. The resulting stream of tuples will be sorted by the values in "group1" and "group2".

```
Pipe[] pipes = Pipe.pipes( lhs, rhs );
Pipe merge = new GroupBy( pipes, new Fields( "group1", "group2" ) );
```

Example 3.3 Merging a Tuple Stream

This example merges two streams ("lhs" and "rhs") into one tuple stream and groups the resulting stream on the fields "group1" and "group2", in the same fashion as the previous example.

CoGroup accepts two or more tuple streams and does not require any common field names. The grouping fields must be provided for each tuple stream.

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

Example 3.4 Joining a Tuple Stream

This example joins two streams ("lhs" and "rhs") on common values. Note that common field names are not required here. Actually, if there were any common field names, the Cascading planner would throw an error as duplicate field names are not allowed.

This is significant because of the nature of joining streams.

The first stage of joining has to do with identifying field names that represent the grouping key for a given stream. The second stage is emitting a new Tuple with the joined values, this includes the grouping values, and the other values.
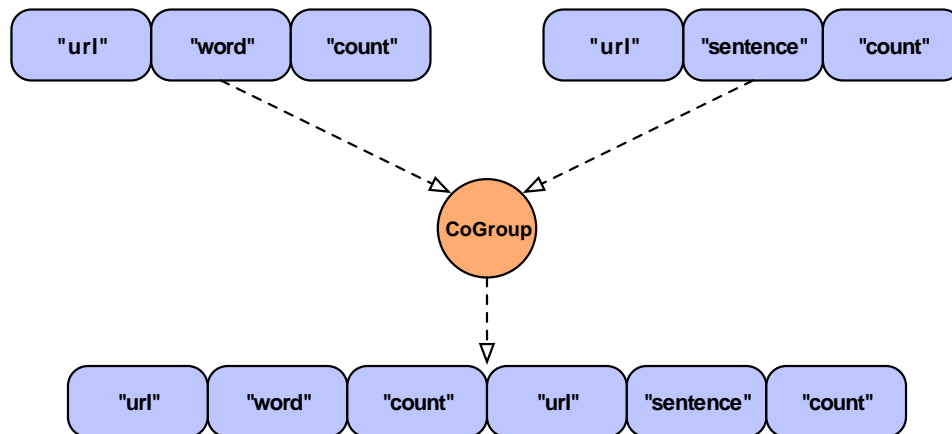
In the above example, we see what "logically" happens during a join. Here we join two streams on the "url" field which happens to be common to both streams. The result is simply two Tuple instances with the same "url" appended together into a new Tuple. In practice this would fail since the result Tuple has duplicate field names. The `CoGroup` pipe has the *declaredFields* argument allowing the developer to declare new unique field names for the resulting tuple.

```
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 3.5 Joining a Tuple Stream with Duplicate Fields
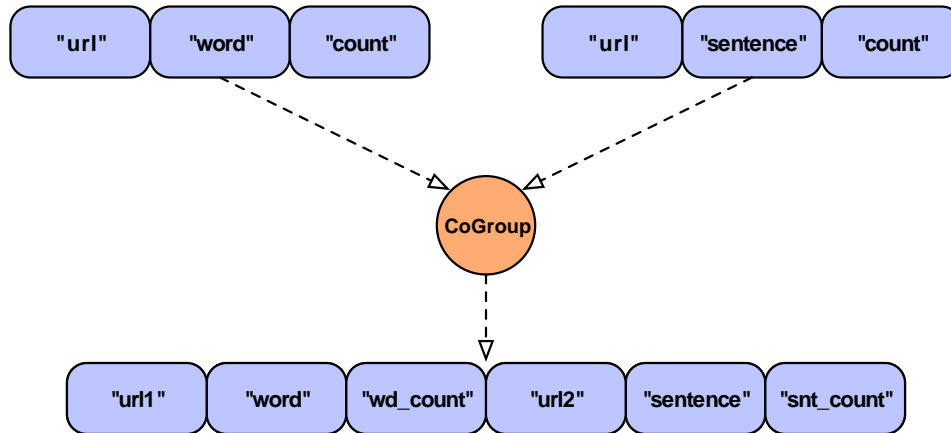
Here we see an example of what the developer could have named the fields so the planner would not fail.

It is important to note that Cascading could just magically create a new Tuple by removing the duplicate grouping fields names so the user isn't left renaming them. In the above example, the duplicate "url" columns could be collapsed into one, as they are the same value. This is not done because field names are a user convenience, the primary mechanism to manipulate Tuples is through positions, not field names. So the result of every Pipe (Each, Every, CoGroup, GroupBy) needs to be deterministic. This gives Cascading a big performance boost, provides a means for sub-assemblies to be built without coupling to any "domain level" concepts (like "first_name", or "url"), and allows for higher level abstractions to be built on-top of Cascading simply.

In the example above, we explicitly set a Joiner class to join our data. The reason `CoGroup` is named "CoGroup" and not "Join" is because joining data is done after all the parallel streams are co-grouped by their common keys. The details are not terribly important, but note that a "bag" of data for every input tuple stream must be created before an join operation can be performed. Each bag consists of all the Tuple instances associated with a given grouping Tuple.
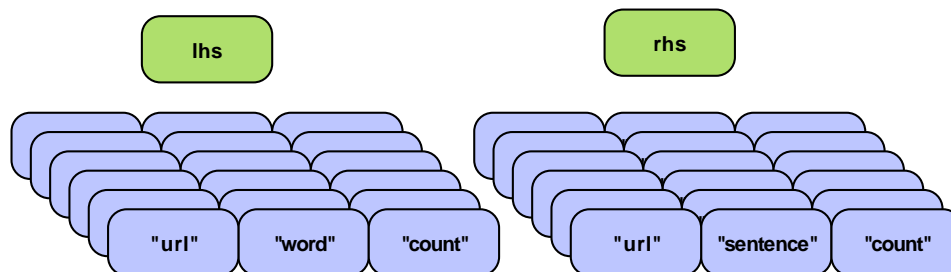
Above we see two bags, one for each tuple stream ("lhs" and "rhs"). Each Tuple in bag is independent but all Tuples in both bags have the same "url" value since we are grouping on "url", from the previous example. A Joiner will match up every Tuple on the "lhs" with a Tuple on the "rhs". An InnerJoin is the most common. This is where each Tuple on the "lhs" is matched with every Tuple on the "rhs". This is the default behaviour one would see in SQL when doing a join. If one of the bags was empty, no Tuples would be joined. An OuterJoin allows for either bag to be empty, and if that is the case, a Tuple full of `null` values would be substituted.
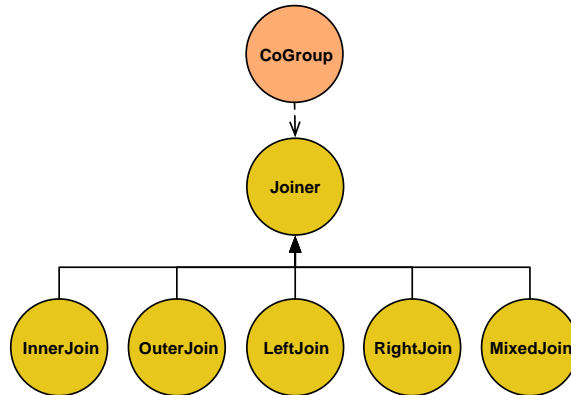
Above we see all supported Joiner types.

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

Using the above simple data sets, we will define each join type where the values are joined on the first position, a numeric value. Note when Cascading joins Tuples, the resulting Tuple will contain all the incoming values. The duplicate common key(s) is not discarded if given. And on outer joins, where there is no equivalent key in the alternate stream, `null` values are used as placeholders.

InnerJoin

An Inner join will only return a joined Tuple if neither bag has is empty.

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

OuterJoin

An Outer join will join if either the left or right 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 fine if the right bag is empty.

```
[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 fine if the left bag is empty.

```
[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, and each pair must be joined differently. See the `cascading.pipe.cogroup.MixedJoin` class for more details.

Custom
   A custom join is where the developer subclasses the `cascading.pipe.cogroup.Joiner` class.

Sorting

By virtue of the Reduce method, in the MapReduce model encapsulated by `GroupBy` and `CoGroup`, all groups of Tuples will be locally sorted by their grouping values. That is, both the `Aggregator` and `Buffer` Operations will receive groups in their natural sort order. But the values associated within those groups are not sorted.

That is, if we sort on 'lastname' with the tuples `[john, doe]` and`[jane, doe]`, the 'firstname' values will arrive in an arbitrary order to the `Aggregator.aggregate()` method.

In the below example we provide sorting fields to the `GroupBy` instance. Now `value1` and `value2` will arrive in their natural sort order (assuming `value1` and `value2` are `java.lang.Comparable`).

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

Example 3.6 Secondary Sorting

If we didn't care about the order of`value2`, would could have left it out of the `sortFields Fields` constructor.

In this 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 3.7 Reversing Secondary Sort Order

Whenever there is an implied sort, during grouping or secondary sorting, a custom `java.util.Comparator` can be supplied to the grouping `Fields` or secondary sort `Fields` to influence the sorting through the `Fields.setComparator()` call.

Creating a custom `Comparator` also allows for non- `Comparable` classes to be sorted and/or grouped on.

Here is a more practical example were 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 3.8 Reverse Order by Time

3.3 Source and Sink Taps

All input data comes from, and all output data feeds to, a `cascading.tap.Tap` instance.

A Tap represents a resource like a data file on the local file system, on a Hadoop distributed file system, or even on Amazon S3. Taps can be read from, which makes it a "source", or written to, which makes it a "sink". Or, more commonly, Taps can act as both sinks and sources when shared between Flows.

All Taps must have a Scheme associated with them. If the Tap is about where the data is, and how to get it, the Scheme is about what the data is. Cascading provides three Scheme classes, `TextLine`,`TextDelimited`, `SequenceFile`, and `WritableSequenceFile`.

TextLine

TextLine reads and writes raw text files and returns Tuples with two field names by default, "offset" and "line". These values are inherited from Hadoop. When written to, all Tuple values are converted to Strings and joined with the TAB character (\t).

TextDelimited

TextDelimited reads and writes character delimited files (csv, tsv, etc). When written to, all Tuple values are converted to Strings and joined with the given character delimiter. This Scheme can optionally handle quoted values with custom quote characters. Further, TextDelimited can coerce each value to a primitive type.

SequenceFile

SequenceFile is based on the Hadoop Sequence file, which is a binary format. When written or read from, all Tuple values are saved in their native binary form. This is the most efficient file format, but being binary, the result files can only be read by Hadoop applications.

WritableSequenceFile

WritableSequenceFile is based on the Hadoop Sequence file, like the SequenceFile Scheme, except 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 will be serialized directly into the sequence file. During reading (sourcing) the key and/or value objects will be deserialized and wrapped in a Cascading Tuple and passed to the downstream pipe assembly.

The fundamental difference behind `TextLine` and `SequenceFile` schemes is that tuples stored in the `SequenceFile` remain tuples, so when read, they do not need to be parsed. So a typical Cascading application

will read raw text files, and parse each line into a `Tuple` for processing. The final Tuples are saved via the `SequenceFile` scheme so future applications can just read the file directly into `Tuple` instances without the parsing step.

It is advised for performance reasons, sequence file compression be enabled via the Hadoop properties. Either block or record based compression can be enabled. See the Hadoop documentation for the available properties and compression types available.

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

Example 3.9 Creating a new Tap

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

The three most common Tap classes used are, Hfs, Dfs, and Lfs. The MultiSourceTap, MultiSinkTap, and TemplateTap are utility Taps.

Lfs

> The `cascading.tap.Lfs` Tap is used to reference local files. Local files are files on the same machine your Cascading application is started. Even if a remote Hadoop cluster is configured, if a Lfs Tap is used as either a source or sink in a Flow, Cascading will be forced to run in "local mode" and not on the cluster. This is useful when creating applications to read local files and import them into the Hadoop distributed file system.

Dfs

> The `cascading.tap.Dfs` Tap is used to reference files on the Hadoop distributed file system.

Hfs

> The `cascading.tap.Hfs` Tap uses the current Hadoop default file system. If Hadoop is configured for "local mode" its default file system will be the local file system. If configured as a cluster, the default file system is likely the Hadoop distributed file system. The Hfs is convenient when writing Cascading applications that may or may not be run on a cluster. Lhs and Dfs subclass the Hfs Tap.

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 an output sink. During runtime, for every Tuple output by the pipe assembly each child tap to the MultiSinkTap will sink the Tuple.

TemplateTap

> The `cascading.tap.TemplateTap` is used to sink tuples into directory paths based on the values in the Tuple. More can be read below inTemplate Taps.

GlobHfs

> The `cascading.tap.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.

Keep in mind Hadoop cannot source data from directories with nested sub-directories, and it cannot write to directories that already exist. But you can simply point the `Hfs Tap` to a directory of data files and they all will be used as input, no need to enumate each individual file into a `MultiSourceTap`.

To get around existing directories, the Hadoop related Taps allow for a `SinkMode` value to be set when constructed.

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

Example 3.10 Overwriting An Existing Resource

Here are all the modes available by the built-in Tap types.

`SinkMode.KEEP`

This is the default behavior. If the resource exists, attempting to write to 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 have the concept of update or append. For example, updating records in a database. It is up to the Tap to decide how to implement its "update" semantics. When Cascading sees the update mode, it knows not to attempt to delete the resource first or to not fail because it already exists.

3.4 Field Algebra

As can be seen above, the `Each` and `Every Pipe` classes provide a means to merge input Tuple values with Operation result Tuple values to create a final output Tuple, which are used as the input to the next `Pipe` instance. This merging is created through a type of "field algebra", and can get rather complicated when factoring in Fields sets, a kind of wildcard for specifying certain field values.

Fields sets are constant values on the `Fields` class and can be used in many places 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.

Fields.RESULTS

The `cascading.tuple.Fields.RESULTS` constant set 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 where it replaces in the input Tuple with the Operation result Tuple in the stream.

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 is a convenience Fields set that allows subsequent Operations to 'step' on the value with a given field name. The current Operation must always use the exact same field names, or the `ARGS` Fields set.

Fields.SWAP

The `cascading.tuple.Fields.SWAP` constant is used as an output selector to swap out Operation arguments with its results. Neither the argument and result field names or 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.

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.

Fields.GROUP

The `cascading.tuple.Fields.GROUP` constant represents all the fields used as grouping values in a previous Group. If there is no previous Group in the pipe assembly, the `GROUP` represents all the current field names.

Fields.VALUES

The `cascading.tuple.Fields.VALUES` constant represent all the fields not used as grouping fields in a previous Group.

Fields.UNKNOWN

The `cascading.tuple.Fields.UNKNOWN` constant is used when Fields must be declared, but how many and their names is unknown. This allows for arbitrarily length Tuples from an input source or some Operation. Use this Fields set with caution.

Below is a reference chart showing common ways to merge input and result fields for the desired output fields. See the section on Each and Every Pipes for details on the different columns and their relationships to the `Each` and `Every` Pipes and Functions, Aggregators, and Buffers.

Input Fields	Argument Selector	Declared Fields	Result Fields	Output Selector	Output Fields	Comments
"line"	ALL	"ip" "time"	"ip" "time"	RESULTS	"ip" "time"	
"line"	"line"	"ip" "time"	"ip" "time"	RESULTS	"ip" "time"	
"line"	ALL	"ip" "time"	"ip" "time"	ALL	"line" "ip" "time"	
"line"	"line"	"ip" "time"	"ip" "time"	ALL	"line" "ip" "time"	
"line"	"line"	"ip" "time"	"ip" "time"	"line" "time"	"line" "time"	
"line"	ALL	UNKNOWN	UNKNOWN	RESULTS	UNKNOWN	
"line"	ALL	UNKNOWN	UNKNOWN	ALL	UNKNOWN	
"ip" "time" "status"	"status"	ARGS	"status"	RESULTS	"status"	
"ip" "time" "status"	"status"	ARGS	"status"	ALL	FAIL	Output selector Allow cause duplicate fie l names.
"ip" "time" "status"	ALL	ARGS	"ip" "time" "status"	RESULTS	"ip" "time" "status"	
"ip" "time" "status"	ALL	ARGS	"ip" "time" "status"	ALL	FAIL	Output selector Allow cause duplicate fie l names.
"ip" "time" "status"	"status"	ARGS	"status"	REPLACE	"ip" "time" "status"	
"date" "time" "status"	"date" "time"	"ts"	"ts"	SWAP	"status" "ts"	

3.5 Flows

When pipe assemblies are bound to source and sink Taps, a `Flow` is created. Flows are executable in the sense that once created they can be "started" and will begin execution on a configured Hadoop cluster.

Think of a Flow as a data processing workflow 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 when the Flow is created, but it must exist when the Flow is executed (unless executed as part of a Cascade, seeCascades).

The most common pattern is to create a Flow from an existing pipe assembly. But there are cases where a MapReduce job 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, or 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

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

Example 3.11 Creating a new Flow

To create a Flow, it must be planned though the FlowConnector object. The `connect()` method is used to create new Flow instances based on a set of sink Taps, source Taps, and a pipe assembly. The example above is quite trivial.

```
// 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" );

Map<String, Tap> sources = new HashMap<String, Tap>();

sources.put( "lhs", lhsSource );
sources.put( "rhs", rhsSource );

Flow flow = new FlowConnector().connect( "flow-name", sources, sink, groupBy );
```

Example 3.12 Binding Taps in a Flow

The example above expands on our previous pipe assembly example by creating source and sink Taps and planning a Flow. Note there are two branches in the pipe assembly, one named "lhs" and the other "rhs". Cascading uses those names to bind the source Taps to the pipe assembly. A HashMap of names and taps must be passed to FlowConnector in order to bind Taps to branches.

Since there is only one tail, the "join" pipe, we don't need to bind the sink to a branch name. Nor do we need to pass the heads of the assembly to the FlowConnector, it can determine the heads of the pipe assembly on the fly. When creating more complex Flows with multiple heads and tails, all Taps will need to be explicitly named, and the proper `connect()` method will need be called.

Configuring Flows

The FlowConnector constructor accepts the `java.util.Property` object so that default Cascading and Hadoop properties can be passed down through the planner to the Hadoop runtime. Subsequently any relevant Hadoop `hadoop-default.xml` properties may be added (`mapred.map.tasks.speculative.execution`, `mapred.reduce.tasks.speculative.execution`, or `mapred.child.java.opts` would be very common).

One property that must 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
FlowConnector.setApplicationJarClass( properties, Main.class );

// or pass in the path to the parent jar
FlowConnector.setApplicationJarPath( properties, pathToJar );

FlowConnector flowConnector = new FlowConnector( properties );
```

Example 3.13 Configuring the Application Jar

More information on packaging production applications can be found inExecuting Processes.

Note the pattern of using a static property setter method (`cascading.flow.FlowConnector.setApplicationJarPath`), other classes that can be used to set properties are `cascading.flow.MultiMapReducePlanner` and `cascading.flow.Flow`.

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.

Skipping Flows

When a Flow participates in a Cascade, the `Flow#isSkip()` method is consulted before calling `Flow#start()` on the flow. By default `isSkip()` returns true if any of the sinks are stale in relation to the Flow sources. Where stale is if they don't exist or the resources are older than the sources.

This behavior is pluggable through the `cascading.flow.FlowSkipStrategy` interface. A new strategy can be set on a `Flow` instance after its created.

FlowSkipIfSinkStale

> The `cascading.flow.FlowSkipIfSinkStale` strategy is the default strategy. Sinks are stale if they don't exist or the resources are older than the sources. If the SinkMode for the sink Tap is REPLACE, then the Tap will be treated as stale.

FlowSkipIfSinkExists

> The `cascading.flow.FlowSkipIfSinkExists` strategy will skip a Flow if the sink Tap exists, regardless of age. If the `SinkMode` for the sink Tap is `REPLACE`, then the Tap will be treated as stale.

Note `Flow#start()` and `Flow#complete()` will not consult the `isSkip()` method and subsequently will always try to start the Flow if called. It is up to user code to call `isSkip()` to decide if the current strategy suggests 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.MapReduceFlow` class should be used. After creating a Hadoop `JobConf` instance, just 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 an Apache licensed set of Java Annotations that identify specific methods on a Class as providing specific life-cycle and dependency functionality. See the Riffle documentation and examples. To use with Cascading, a Riffle annotated instance must be passed to the `cascading.flow.FlowProcess` constructor method. The resulting FlowProcess instance can be used like any other Flow instance.

Since many algorithms need to have 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 a Flows and Cascades in a parent Flow which in turn can participate in a Cascade.

3.6 Cascades

A Cascade allows multiple Flow instances to be executed as a single logical unit. If there are dependencies between the Flows, they will be executed in the correct order. Further, Cascades act like ant build or Unix "make" files. When run, a Cascade will only execute Flows that have stale sinks (output data that is older than the input data), by default.

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

Example 3.14 Creating a new Cascade

When passing Flows to the CascadeConnector, order is not important. The CascadeConnector will automatically determine what the dependencies are between the given Flows and create a scheduler that will start each flow as its

data sources become available. If two or more Flow instances have no dependencies, they will be submitted together so they can execute in parallel.

For more information, see the section onTopological Scheduling.

If an instance of `cascading.flow.FlowSkipStrategy` is given to an `Cascade` instance via the `Cascade#setFlowSkipStrategy()` method, it will be consulted for every Flow instance managed by the Cascade, all skip strategies on the Flow instances will be ignored. For more information on skip strategies, seeSkipping Flows.

4. Executing Processes

4.1 Introduction

Cascading requires Hadoop to be installed and correctly configured. 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/.

4.2 Building

Cascading ships with a handful of jars.

cascading-1.2.x.jar

    all relevant Cascading class files and libraries, with a Hadoop friendly `lib` folder containing all third-party dependencies

cascading-core-1.2.x.jar

    all Cascading Core class files, should be packaged with `lib/*.jar`

cascading-xml-1.2.x.jar

    all Cascading XML module class files, should be packaged with `lib/xml/*.jar`

cascading-test-1.2.x.jar

    all Cascading unit tests. If writing custom modules for cascading, sub-classing `cascading.CascadingTestCase` might be helpful

Cascading will run with Hadoop in its default 'local' or 'stand alone' mode, or configured as a distributed cluster.

When used on a cluster, a Hadoop job Jar must be created with Cascading jars and dependent thrid-party jars in the job jar `lib` directory, per the Hadoop documentation.

```
<!-- Common ant build properties, included here for completeness -->
<property name="build.dir" location="${basedir}/build"/>
<property name="build.classes" location="${build.dir}/classes"/>


<!-- Cascading specific properties -->
<property name="cascading.home" location="${basedir}/../cascading"/>
<property file="${cascading.home}/version.properties"/>
<property name="cascading.release.version" value="x.y.z"/>
<property name="cascading.filename.core"
          value="cascading-core-${cascading.release.version}.jar"/>
<property name="cascading.filename.xml"
          value="cascading-xml-${cascading.release.version}.jar"/>
<property name="cascading.libs" value="${cascading.home}/lib"/>
<property name="cascading.libs.core" value="${cascading.libs}"/>
<property name="cascading.libs.xml" value="${cascading.libs}/xml"/>


<condition property="cascading.path" value="${cascading.home}/"
           else="${cascading.home}/build">
  <available file="${cascading.home}/${cascading.filename.core}"/>
</condition>


<property name="cascading.lib.core"
          value="${cascading.path}/${cascading.filename.core}"/>
<property name="cascading.lib.xml"
          value="${cascading.path}/${cascading.filename.xml}"/>
```

Example 4.1 Sample Ant Build - Properties

```
<!--
  A sample target to jar project classes and Cascading
  libraries into a single Hadoop compatible jar file.
 -->

<target name="jar" description="creates a Hadoop ready jar w/dependencies">

  <!-- copy Cascading classes and libraries -->
  <copy todir="${build.classes}/lib" file="${cascading.lib.core}"/>
  <copy todir="${build.classes}/lib" file="${cascading.lib.xml}"/>
  <copy todir="${build.classes}/lib">
    <fileset dir="${cascading.libs.core}" includes="*.jar"/>
    <fileset dir="${cascading.libs.xml}" includes="*.jar"/>
  </copy>

  <jar jarfile="${build.dir}/${ant.project.name}.jar">
    <fileset dir="${build.classes}"/>
    <fileset dir="${basedir}" includes="lib/"/>
    <manifest>
      <!-- the project Main class, by default assumes Main -->
      <attribute name="Main-Class" value="${ant.project.name}/Main"/>
    </manifest>
  </jar>

</target>
```

Example 4.2 Sample Ant Build - Target

The above Ant snippets can be used in your project to create a Hadoop jar for submission on your cluster. Again, all Hadoop applications that are intended to be run in a cluster must be packaged with all third-party libraries in a directory named `lib` in the final application Jar file, regardless if they are Cascading applications or raw Hadoop MapReduce applications.

Note, the snippets above is only intended to show how to include Cascading libraries, you still need to compile your project into the build.classes path.

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.

```
Properties properties = new Properties();

// pass in the class name of your application
```

```
// this will find the parent jar at runtime
FlowConnector.setApplicationJarClass( properties, Main.class );

// or pass in the path to the parent jar
FlowConnector.setApplicationJarPath( properties, pathToJar );

FlowConnector flowConnector = new FlowConnector( properties );
```

Above we see how to set the same property two ways. First via the `setApplicationJarClass()` method, and via the `setApplicationJarPath()` method. 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 will be pushed to the cluster, not the parent application jar.

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

4.4 Executing

Running a Cascading application is exactly the same as running any Hadoop application. After packaging your application into a single jar (seeBuilding Cascading Applications), 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 4.3 Running a Cascading Application

If the configuration scripts in `$HADOOP_CONF_DIR` are configured to use a cluster, the Jar will be 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, likely one of the internal libraries (found in the lib folder) will be pushed to the cluster instead and `ClassNotFoundExceptions` will be thrown.

5. Using and Developing Operations

5.1 Introduction

To use Cascading, it is not strictly necessary to create custom Operations. There are a number of Operations in the Cascading library that can be combined into very robust applications. In the same way you can chain**sed**,**grep**, **sort**,**uniq**, **awk**, etc in Unix, you can chain existing Cascading operations. But developing customs Operations is very simple in Cascading.

There are four kinds of Operations: `Function`,`Filter`, `Aggregator`, and `Buffer`.
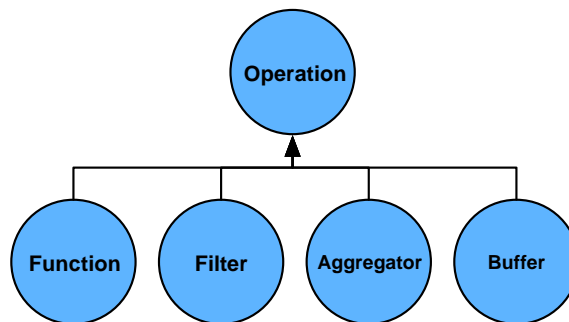
All Operations operate on an input argument Tuple and all Operations other than `Filter` may return zero or more Tuple object results. That is, a `Function` can parse a string and return a new Tuple for every value parsed out (one Tuple for each 'word'), or it may create a single Tuple with every parsed value as an element in the Tuple object (one Tuple with "first-name" and "last-name" fields).

In practice, a `Function` that returns no results is a`Filter`, but the `Filter` type has been optimized and can be combined with "logical" filter Operations like `Not`,`And`, `Or`, etc.

During runtime, Operations actually receive arguments as an instance of the TupleEntry object. The TupleEntry object holds both an instance of `Fields` and the current `Tuple` the `Fields` object defines fields for.

All Operations, other than`Filter`, must declare result Fields. For example, if a `Function` was written to parse words out of a String and return a new Tuple for each word, this `Function` must declare that it intends to return a Tuple with one field named "word". If the `Function` mistakenly returns more values in the Tuple other than a 'word', the process will fail. Operations that do 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 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 result Tuple.

Operations, by default, are "safe". Safe Operations can execute safely multiple times on the same Tuple multiple times, that is, it has no side-effects, it is idempotent. If an 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 single argument `Tuple`, and may return zero or more result Tuples.

A `Function` may only be used with a `Each` pipe which may follow any other pipe type.

To create a custom `Function`, subclass the class `cascading.operation.BaseOperation` and implement the interface `cascading.operation.Function`. Because `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

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

Functions must accept 1 or more values in a Tuple as arguments, by default they will accept any number (`Operation.ANY`) of values. Cascading will verify that the number of arguments selected match the number of arguments expected during the planning phase.

Functions may optionally declare the field names they return, by default `Functions` declare `Fields.UNKNOWN`.

Both declarations 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 fully functional `Function` that accepts two values in the argument Tuple, adds them together, and returns the result in a new Tuple.

The first constructor assumes a default field name this function will return, but it is a best practice to always give the user the option to override the declared field names to prevent any field name collisions that would cause the planner to fail.

5.3 Filter

A `Filter` expects a single argument Tuple and returns a boolean value stating whether or not the current Tuple in the tuple stream should be discarded.

A `Filter` may only be used with a `Each` pipe, and it may follow any other 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 filterCall )
    {
    // get the arguments TupleEntry
    TupleEntry arguments = filterCall.getArguments();

    // initialize the return result
    boolean isRemove = false;

    // test the argument values and set isRemove accordingly

    return isRemove;
    }
  }
```

Example 5.3 Custom Filter

Filters should declare the number of argument values they expect.

Filters must accept 1 or more values in a Tuple as arguments, by default they will accept any number (`Operation.ANY`) of values. Cascading will verify the number of arguments selected match the number of arguments expected.

The number of arguments declarations 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 filterCall )
    {
    // get the arguments TupleEntry
    TupleEntry arguments = filterCall.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 5.4 String Length Filter

The example above implements a fully functional `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 set of argument Tuples in the same grouping, and may return zero or more result Tuples.

An `Aggregator` may only be used with an `Every` pipe, and it may only follow a `GroupBy`,`CoGroup`, or another `Every` pipe type.

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
```

Example 5.5 Custom Aggregator

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

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

Aggregators must accept 1 or more values in a Tuple as arguments, by default they will accept any number (`Operation.ANY`) of values. Cascading will verify the number of arguments selected match the number of arguments expected.

Aggregators may optionally declare the field names they return, by default `Aggregators` declare `Fields.UNKNOWN`.

Both declarations 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 5.6 Add Tuples Aggregator

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

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

The example above implements a fully functional `Aggregator` that accepts one value in the argument Tuple, adds all these argument Tuples in the current grouping, and returns the result as a new Tuple.

The first constructor assumes a default field name this `Aggregator` will return, but it is a best practice to always give the user the option to override the declared field names to prevent any field name collisions that would cause the planner to fail.

5.5 Buffer

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

The `Buffer` is very similar to an `Aggregator` except 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, and is best used for operations that need greater visibility to the previous and next elements in the stream. For example, smoothing a series of time-stamps where there are missing values.

An `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 5.7 Custom Buffer

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

Buffers must accept 1 or more values in a Tuple as arguments, by default they will accept any number (Operation.ANY) of values. Cascading will verify the number of arguments selected match the number of arguments expected.

Buffers may optionally declare the field names they return, by default Buffers declare Fields.UNKNOWN.

Both declarations 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 5.8 Average Buffer

The example above implements a fully functional buffer that accepts one value in 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 assumes a default field name this `Buffer` will return, but it is a best practice to always give the user the option to override the declared field names to prevent any field name collisions that would cause the planner to fail.

Note this example is somewhat fabricated, in practice a `Aggregator` should be implemented to compute averages. A `Buffer` would be better suited for "running averages" across very large spans, for example.

5.6 Operation and BaseOperation

In all 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`, 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, a `java.util.regex.Matcher` may need to be initialized and used in a thread-safe way. Or a remote connection may need to be opened and eventually closed. But for performance reasons, the operation should not create/destroy the connection for each Tuple or every Tuple group that passes through.

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. `prepare()` is called before any argument Tuple is passed in, and `cleanup()` 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. The "context" is user defined and is the same mechanism used by the `Aggregator` operation, except the `Aggregator` is also given the opportunity to initialize and destroy its context via the `start()` and `complete()` methods.

If a "context" object is used, its type should be declared in the sub-class class declaration using the Java Generics notation.

6. Advanced Processing

6.1 SubAssemblies

Cascading SubAssemblies are reusable pipe assemblies that are linked into larger pipe assemblies. Think of them as subroutines in a programming language. The help organize complex pipe assemblies and allow for commonly used pipe assemblies to be packaged into libraries for inclusion by other users.

To create a SubAssembly, the `cascading.pipe.SubAssembly` class must be subclassed.

```
public class SomeSubAssembly extends SubAssembly
  {
  public SomeSubAssembly( Pipe lhs, Pipe rhs )
    {
    // continue assembling against lhs
    lhs = new Each( lhs, new SomeFunction() );
    lhs = new Each( lhs, new SomeFilter() );

    // continue assembling against lhs
    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 6.1 Creating a SubAssembly

In the above example, we pass in via the constructor pipes we wish to continue assembling against, and the last line we register the '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 6.2 Using a SubAssembly

Above we see how natural it is to include a SubAssembly into a new pipe assembly.

If we had a SubAssembly that represented a split, that is, had two or more tails, we could use the `getTails()` method to get at the array of "tails" set internally by the `setTails()` method.

```
public class SplitSubAssembly extends SubAssembly
  {
  public SplitSubAssembly( Pipe pipe )
    {
    // continue assembling against lhs
    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 6.3 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 6.4 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, each branch tail must be passed to the `setTails()` method, and the `getTails()` method should be called to get a handle to the correct branch to pass to the next `Pipe` instances.

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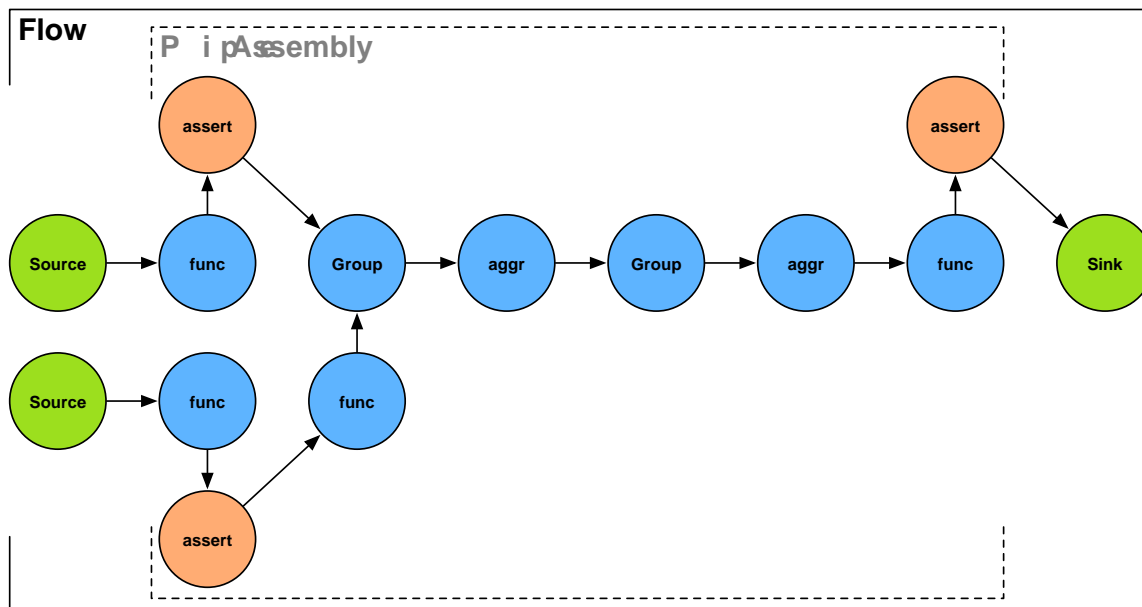
Stream assertions are simply a mechanism to 'assert' that one or more values in a tuple stream meet certain criteria. This is similar to the Java language 'assert' keyword, or a unit test. An example would be 'assert not null' or 'assert matches'.

Assertions are treated like any other function or aggregator in Cascading. They are embedded directly into the pipe assembly by the developer. If an assertion fails, the processing stops, by default. Alternately they can trigger a Failure Trap.

As with any test, sometimes they are wanted, and sometimes they are unnecessary. Thus stream assertions are embedded as either 'strict' or 'validating'.

When running a tests against regression data, it makes sense to use strict assertions. This regression data should be small and represent many of the edge cases the processing assembly must support robustly. When running tests in staging, or with data that may vary in quality since it is from an unmanaged source, using validating assertions make much sense. Then there are obvious cases where assertions just get in the way and slow down processing and it would be nice to just bypass them.

During runtime, Cascading can be instructed to plan out strict, validating, or all assertions before building the final MapReduce jobs via the MapReduce Job Planner. And they are truly planned out of the resulting job, not just switched off, providing the best performance.

This is just one feature of lazily building MapReduce jobs via a planner, instead of hard coding them.

```
// 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 6.5 Adding Assertions

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

```
Properties properties = new Properties();

// removes all assertions from the Flow
FlowConnector.setAssertionLevel( properties, AssertionLevel.NONE );

FlowConnector flowConnector = new FlowConnector( properties );

Flow flow = flowConnector.connect( source, sink, assembly );
```

Example 6.6 Planning Out Assertions

To configure the planner to remove some or all assertions, a property must be set via the `FlowConnector#setAssertionLevel()` method. `AssertionLevel.NONE` removes all assertions. `AssertionLevel.VALID` keeps `VALID` assertions but removes `STRICT` ones. And `AssertionLevel.STRICT` keeps all assertions, which is the planner default value.

6.3 Failure Traps

Flow
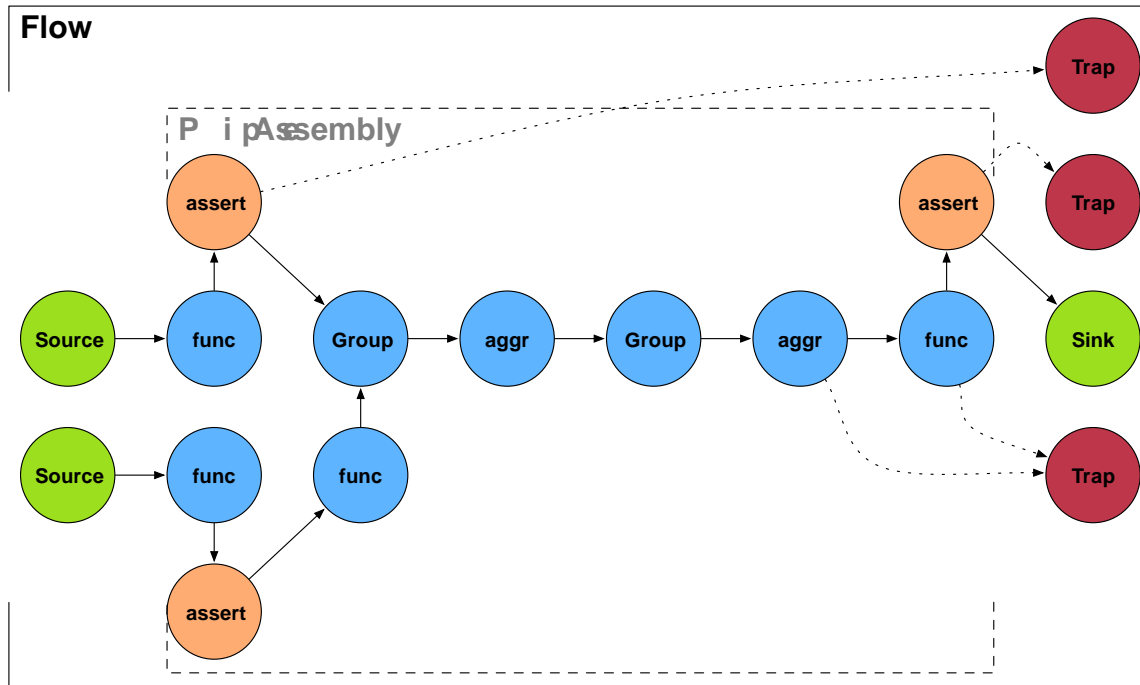
Pipe Assembly

Failure Traps are the same as a Tap sink (opposed to a source), except being bound to a particular tail element of the pipe assembly, traps can be bound to intermediate pipe assembly segments, like to a Stream Assertion.

Whenever an operation fails and throws an exception, and there is an associated trap, the offending Tuple will be 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, the cluster continues working.

But software fault tolerance 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 so attractive, but low fidelity applications (like web page indexing) 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)" );
assembly = new Each( assembly, new Fields("method"),
                     AssertionLevel.STRICT, matchesAll );

// ...some more useful pipes here

Map<String,Tap> traps = new HashMap<String,Tap>();

traps.put( "assertions", trap );

FlowConnector flowConnector = new FlowConnector();
Flow flow =
  flowConnector.connect( "log-parser", source, sink, traps, assembly );
```

Example 6.7 Setting Traps

In the above example, we bind our trap Tap to the pipe assembly segment named "assertions". Note how we can name branches and segments by using a single `Pipe` instance and it applies to all subsequent `Pipe` instances.

Note

Traps are for exceptional cases, in the same way Java Exception handling is not for application flow control, thus traps are not a means to filter some data into other locations. Applications that need to filter good and bad data should do so explicitly.

6.4 Event Handling

Each Flow, has the ability to execute callbacks via an event listener. This is very useful when external application need to be notified that a Flow has completed.

A good example is when running Flows on an Amazon EC2 Hadoop cluster. After the Flow is completed, a SQS event can be sent notifying another application it can now fetch the job results from S3. In tandem, it can start the process of shutting down the cluster if no more work is queued up for it.

Flows support event listeners through the `cascading.flow.FlowListener` interface. The FlowListener interface supports four events,`onStarting`, `onStopping`,`onCompleted`, and `onThrowable`.

onStarting

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

onStopping

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

onCompleted

> The onCompleted event is fired when a Flow instance has completed all work whether if was success or failed. If there was a thrown exception, 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()` method.

FlowListeners are useful when external systems must be notified when a Flow has completed or failed.

6.5 Template Taps

The `TemplateTap Tap` class provides a simple means to break large datasets into smaller sets based on values in the dataset. Typically this is called 'binning' the data, where each 'bin' of data is named after values shared by the data in that bin. For example, organizing log files by month and year.

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

String template = "%s-%s"; // dirs named "year-month"
Tap months = new TemplateTap( tap, template, SinkMode.REPLACE );
```

In the above example, we construct a parent `Hfs Tap` and pass it to the constructor of a `TemplateTap` instance along with a String format 'template'. This format template is populated in the order values are declared via the `Scheme` class. If more complex path formatting is necessary then you may subclass the `TemplateTap`.

Note that you can only create sub-directories to bin data into. Hadoop must still write 'part' files into each bin directory.

One last thing to keep in mind is whether or not 'binning' happens during the Map or Reduce phase. By doing a `GroupBy` on the values that will be used to populate the template, binning will happen during the Reduce phase and likely scale much better if there are a very large number of unique grouping keys.

6.6 Scripting

Cascading was designed with scripting in mind. Since it is just an API, any Java compatible scripting language can import and instantiate Cascading classes and create pipe assemblies, flows, and execute those flows.

And if the scripting language in question supports Domain Specific Language (DSL) creation, the user can create her own DSL to handle common idioms.

See the Cascading website for publicly available scripting language bindings.

6.7 Custom Taps and Schemes

Cascading was designed to be easily configured and enhanced by developers. Besides allowing for custom Operations, developers can provide custom `Tap` and `Scheme` types so applications can connect to system external to Hadoop.

A Tap represents something "physical", like a file or a database table. Subsequently Tap implementations are responsible for life cycle issues around the resource they represent, like tests for existence, or deleting.

A Scheme represents a format or representation, like a text format for a file, or columns in a table. Schemes are responsible for converting the Tap managed resources proprietary format to and from a `cascading.tuple.Tuple` instance.

Unfortunately creating custom Taps and Schemes can be an involved process and requires some knowledge of Hadoop and the Hadoop FileSystem API. Most commonly, the `cascading.tap.Hfs` class can be subclassed if a new file system is to be supported, assuming passing a fully qualified URL to the `Hfs` constructor isn't sufficient (the `Hfs` tap will look up a file system based on the URL scheme via the Hadoop FileSystem API).

Delegating to the Hadoop FileSystem API is not a strict requirement, but the developer will need to implement a Hadoop `org.apache.hadoop.mapred.InputFormat` and/or `org.apache.hadoop.mapred.OutputFormat` so that Hadoop knows how to split and handle the incoming/ outgoing data. The custom `Scheme` is responsible for setting `InputFormat` and `OutputFormat` on the `JobConf` via the `sinkInit` and `sourceInit` methods.

For examples on how to implement a custom Tap and Scheme, see the Cascading Modules [???] page for samples.

6.8 Custom Types and Serialization

The `Tuple` class is a generic container for all `java.lang.Object` instances (1.0 required all objects be of type `java.lang.Comparable`). Subsequently 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 any Class that isn't a primitive value or a Hadoop `Writable` type will need to have 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 registering a new serialization helper or to create `Writable` types. Cascading will automatically inherit any registered serialization implementations.

During serialization and deserialization of `Tuple` instances that contain custom types, the Cascading `Tuple` serialization framework will need to store the class name (as a `String`) before serializing the custom object. This can be very space inneficient. 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 named tokens, and one of Class name strings. Both arrays must be the same size, and no token can be less than 128 (the first 128 values are for internal use).

During serialization and deserialization, the token values are used instead of the `String` Class names 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 Cascading will natively serialize/deserialize all primitives and byte arrays (`byte[]`). It also uses the token 127 for the Hadoop `BytesWritable` class.

6.9 Partial Aggregation instead of Combiners

Cascading does not support the so called MapReduce Combiners. Combiners are very powerful in that they reduce the IO between the Mappers and Reducers. Why send all your Mapper to data to Reducers when you can compute some values Map side and combine them in the Reducer. But Combiners are limited to Associative and Commutative functions only, like 'sum' and 'max'. And in order to work, values emitted from the Map task must be serialized, sorted (deserialized and compared), deserialized again and operated on, where again the results are serialized and sorted. Combiners trade CPU for gains in IO.

Cascading takes a different approach by providing a mechanism to perform partial aggregations Map side and also combine them Reduce side. But Cascading chooses to trade Memory for IO gains by caching values (up to a threshold). This approach bypasses the unnecessary serialization, deserialization, and sorting steps. It also allows for any aggregate function to be implemented, not just Associative and Commutative ones.

Cascading has a few built in partial aggregate operations, actually these "operations" are SubAssemblies. Further, they are implementations of the AggregateBy SubAssembly.

Using partial aggregate operations is quite easy, they are actually less verbose than using 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 6.8 Using a SumBy

To compose multiple partial aggregate operations, things work slightly 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" );
SumBy sumBy = new SumBy( valueField, sumField, long.class );

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

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

Example 6.9 Composing partials with AggregateBy

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

7. Built-In Operations

7.1 Identity Function

The `cascading.operation.Identify` function is used to "shape" a tuple stream. Here are some common patterns.

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 );
pipe = new Each( pipe, new Fields( "ip", "method" ), 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. Additionally `Fields.RESULTS` can be left off as it is the default for the `Every` pipe.

```
// 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 arguments selected 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 above example, if there were more fields than "ip" and "method", it would work fine, all the extra fields would be discarded. If the same was 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.

```
// 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, but return it along with an input Tuple field as the result.

```
// 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`, respectively.

```
// 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 `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"
```

7.2 Debug Function

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

Below we insert a `Debug` operation at the `VERBOSE` level, but configure the planner to remove all `Debug` operations from the resulting `Flow`.

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

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

Properties properties = new Properties();

// tell the planner remove all Debug operations
FlowConnector.setDebugLevel( properties, DebugLevel.NONE );
// ...
FlowConnector flowConnector = new FlowConnector( properties );

Flow flow = flowConnector.connect( "debug", source, sink, assembly );
```

7.3 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.

7.4 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.

7.5 Text Functions

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

FieldJoiner

> The `cascading.operation.text.FieldJoiner` function joins all the values in a Tuple with a given delimiter and stuffs the result into a new field.

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.

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. By default it emits 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
```

> Above we convert an Apache log style date-time field into a `long` timestamp.

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. And uses the `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
```

> Above we convert a `long` timestamp ("ts") to a date String.

7.6 Regular Expression Operations

RegexSplitter

> The `cascading.operation.regex.RegexSplitter` function will split an argument value by a regex pattern String. Internally, this function uses `java.util.regex.Pattern#split()`, thus behaves accordingly. By default this function splits on the TAB character ("\t"). If a known 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 will be discarded, see `java.util.regex.Pattern#split( input, limit )` for more information.

RegexParser

The `cascading.operation.regex.RegexParser` function is used to extract a regular expression matched value from an incoming argument value. If the regular expression is sufficiently complex, and int array may be provided which specifies which regex groups should be returned into 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"
```

Above, we parse an Apache log "line" into its parts. Note the int[] groups array starts at 1, not 0. Group 0 is the whole group, so if included the first field would be a copy of "line" and not "ip".

RegexReplace

The `cascading.operation.regex.RegexReplace` function is used to replace a regex matched value with a replacement value. It maybe used in a "replace all" or "replace first" mode. See `java.util.regex.Matcher#replaceAll()` and `java.util.regex.Matcher#replaceFirst()` methods.

```
// incoming -> "line"

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

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

Above we replace all adjoined white space characters with a single space character.

RegexFilter

The `cascading.operation.regex.RegexFilter` function will apply a regular expression pattern String against every input Tuple value and filter the Tuple stream accordingly. By default, Tuples that match the given pattern are kept, and Tuples that do not match are filtered out. This can be changed by setting "removeMatch" to `true`. Also, by default, the whole Tuple is matched against the given pattern String (TAB delimited). If "matchEachElement" is set to `true`, the pattern is applied to each Tuple value individually. 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"
```

Above we keep all lines where the "ip" address starts with "68.".

RegexGenerator

    The `cascading.operation.regex.RegexGenerator` function will emit a new Tuple for every matched regular expression group, instead of a Tuple with every group as a value.

```
// 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 will emit 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.

7.7 Java Expression Operations

Cascading provides some support for dynamically compiled Java expression to be used as either `Functions` or `Filters`. This functionality is provided by the Janino embedded compiler. Janino and its documentation can be found on its website, http://www.janino.net/. But in short, an Expression is a single line of Java, for example `a + 3 * 2` or `a < 7`. The first would resolve to some number, the second to a boolean value. Where a and b are field names passed in as Tuple arguments to the Operation. Janino will compile this expression into byte code giving compiled code processing speeds.

ExpressionFunction

    The `cascading.operation.expression.ExpressionFunction` function dynamically resolves a given expression using argument Tuple values as inputs to the fields specified 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 form our expression. Note we must declare the type of every input Tuple value so the expression compiler knows how to treat the variables in the expression.

ExpressionFilter

The `cascading.operation.expression.ExpressionFilter` filter dynamically resolves a given expression using argument Tuple values as inputs to the fields specified in the expression. Any Tuple that returns true for the given expression will be 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"
```

Above, every line in the Apache log that does not have a "200" status will be filtered out. Notice that the "status" would be a String in this example if it was emitted from a RegexParser, if so the ExpressionFilter will coerce the value from a String to an `int` for the comparison.

7.8 XML Operations

All XML Operations are kept in a module other than core, so can be included in a Cascading application by including the `cascading-xml-x.y.z.jar` in the project. This module has one dependency, the TagSoup library, which allows for HTML and XML "tidying". More about TagSoup can be read on its website,http://home.ccil.org/~cowan/ XML/tagsoup/.

XPathParser

The `cascading.operation.xml.XPathParser` function will extract a value from the passed Tuple argument into a new Tuple field value. One Tuple value for every given XPath expression will be created. This function effectively converts an XML document into a table. If the returned value of the expression is a`NodeList`, only the first `Node` is used. The `Node` is converted to a new XML document and converted to a String. If only the text values are required, search on the `text()` nodes, or consider using XPathGenerator to handle multiple `NodeList` values.

XPathGenerator

The `cascading.operation.xml.XPathGenerator`function is a generator function that will emit a new Tuple for every `Node` returned by the given XPath expression.

XPathFilter

    The `cascading.operation.xml.XPathFilter` filter will filter out a Tuple if the given XPath expression returns `false`. Set the removeMatch parameter to `true` if the filter should be reversed.

TapSoupParser

    The `cascading.operation.xml.TagSoupParser` function uses the Tag Soup library to convert incoming HTML to clean XHTML. Use the `setFeature( feature, value )` method to set TagSoup specific features (as documented on the TagSoup website listed above).

7.9 Assertions

Cascading Stream Assertions are used to build robust reusable pipe assemblies. They can be planned out of a Flow instance during runtime. For more information see the section onStream Assertions. Below we describe the Assertions available in the core library.

AssertEquals

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

AssertNotEquals

    The `cascading.operation.assertion.AssertNotEquals` Assertion asserts 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 is `.equals()` to 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 whole argument Tuple by joining each individual element of the Tuple with a TAB character (\t).

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 value in the argument Tuple is not a `null` value.

AssertNull

    The `cascading.operation.assertion.AssertNull` Assertion asserts that every value in the argument Tuple is a `null` value.

AssertSizeEquals

    The `cascading.operation.assertion.AssertSizeEquals` Assertion asserts that the current Tuple in the tuple stream is exactly the given size. On evaluation, `Tuple#size()` is called (note Tuples may hold `null` values).

AssertSizeLessThan

    The `cascading.operation.assertion.AssertSizeLessThan` Assertion asserts that the current Tuple in the stream has a size less than (<) the given size. On evaluation, `Tuple#size()` is called (note Tuples may hold `null` values).

AssertSizeMoreThan

    The `cascading.operation.assertion.AssertSizeMoreThan` Assertion asserts that the current Tuple in the stream has a size more than (>) the given size. On evaluation, `Tuple#size()` is called (note Tuples may hold `null` values).

AssertGroupSizeEquals

    The `cascading.operation.assertion.AssertGroupSizeEquals` Group Assertion asserts that the number of items in the current grouping is equal (==) 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 more 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.

7.10 Logical Filter Operators

The logical `Filter` operators allows the user to assemble more complex filtering to be used in a single Pipe, instead of chaining multiple Pipes together to get the same effect.

And

    The `cascading.operation.filter.And` Filter will logically 'and' the results of the constructor provided `Filter` instances. Logically, if `Filter#isRemove()` returns `true` for all given instances, this filter will return`true`.

Or

    The `cascading.operation.filter.Or` Filter will logically 'or' the results of the constructor provided `Filter` instances. Logically, if `Filter#isRemove()` returns `true` for any of the given instances, this filter will return`true`.

Not

The `cascading.operation.filter.Not` Filter will logically 'not' (negation) the results of the constructor provided `Filter` instances. Logically, if `Filter#isRemove()` returns `true` for the given instance, this filter will return the opposite,`false`.

Xor

The `cascading.operation.filter.Xor` Filter will logically 'xor' (exclusive or) the results of the constructor provided `Filter` instances. Logically, if `Filter#isRemove()` returns `true` for all given instances, or returns `false` for all given instances, this filter will return`true`. Note that Xor can only be applied to two values.

```
// 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 7.1 Combining Filters

Above, we are "and-ing" the two filters. Both must be satisfied for the data to pass through this one Pipe.

8. Best Practices

8.1 Unit Testing

Testing Operations, pipe-assemblies, and applications is a must. The `cascading.CascadingTestCase` provides a number of 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, each offering extra flexibility. All of them will read the sink Tap and validate it is the correct length, have the correct Tuple size, and if the values match a given regular expression pattern.

The `cascading.ClusterTestCase` can be used if you want to launch an embedded Hadoop cluster inside your TestCase.

Make sure `cascading-test-x.y.z.jar` is in your testing class-path in order to use these helper classes.

8.2 Flow Granularity

Even though having one large `Flow` may result in a slightly more efficient execution plan, it is much more modular and flexible to give smaller Flows well defined responsibilities and to hand all the dependent Flow instances to a `Cascade` for execution as a single unit. Using the `TextDelimited Scheme` between `Flow` instances also provides a means to hand intermediate data off to other systems for reporting or QA with minimal penalty while remaining compatible with other tools.

8.3 SubAssemblies, not Factories

When developing your applications, use `SubAssembly` sub-classes, not "factory" methods. This way the code is much easier to read and to test.

The funny thing is that `Object` constructors are "factories", so there isn't much reason to build frameworks to duplicate what a constructor already does. Of course there are exceptions, but in practice they are rare when you can use a `SubAssembly`.

8.4 Give SubAssemblies Logical Responsibilities

SubAssembies provide a very convenient means to co-locate like responsibilities into a single place. For example, have 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 if a given `Tuple` should be discarded or marked as bad.

Further, in your unit tests, you can create an `TestAssertionsSubAssembly`, that just inlines various `ValueAssertions` and `GroupAssertions`. Inlining Assertions directly in your SubAssemblies is also very important, but sometimes it makes sense to have more tests outside of the business logic.

8.5 Java Operators in Field Names

There are a number of Operations in Cascading that will compile and apply Java expressions on the fly, see `ExpressionFunction` and `ExpressionFilter` for examples. In these expressions, Operation argument field names are used as variable in the expression. When creating field names, be conscious of the fact that if they are used in an expression, some characters will cause compilation errors. For example, "first-name" is a valid field name for use with Cascading, but this expression,`first-name.trim()`, will fail.

8.6 Debugging Planner Failures

Oftentimes the `FlowConnector` will fail when attempting to plan a`Flow`. If the exception message given by `PlannerException` is vague, use the method `PlannerException.writeDOT()` to export a text 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.

Also note you can create a DOT file from a `Flow` as well via `Flow.writeDOT()`.

8.7 Optimizing Joins

When joining two streams via a CoGroup Pipe, attempt to place the largest of the streams in the left most argument to the CoGroup. Joining multiple streams requires some accumulation of values before the join operator can begin, but the left most stream will not be accumulated. This should improve the performance of most joins.

8.8 Debuging Streams

When creating complex assemblies it is safe to embed `Debug` operations (see`Debug` Function) at appropriate debug levels where appropriate. Use the planner to remove them at runtime for production and staging runs to prevent them from using unnecessary resources.

8.9 Handling Good and Bad Data

It is very common when processing raw data streams to encounter data that is corrupt or malformed in some way. This may be because bad content was fetched off the web via a crawler/fetcher upstream. Or a bug leaked into a browser widget that sends user behavior information back for analysis. Whatever the use-case, there is likely a set of rules that govern when to identify and choose to keep or discard a questionable record.

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

Instead 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/bad boolean value. Optionally, set a reason field as to why the Tuple was marked bad.

8.10 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 your Operation will be called from a single thread in a JVM, future version of Hadoop could execute the same operation from multiple threads.

To maintain state across `Operation` calls, create and initialize a "context" object that is maintained by the appropriate `OperationCall` (`FilterCall`,`FunctionCall`, `AggregatorCall`, and `BufferCall`). In the example above, store an Integer 0 in the `AggregatorCall` passed to the `Aggregator.start()` method and increment it in the `Aggregator.aggregate()` method.

8.11 Custom Types

It is generally frowned upon to pass a custom class through a Tuple stream. One one hand this increases coupling of custom Operations to a particular type, and it removes opportunities for reducing the amount of data that passes over the network (or is serialized/deserialized).

To overcome the first objection, with every custom type with multiple instance fields, attempt to provide Functions that can promote a value from the custom object to a position in a Tuple or demote the Tuple value to a particular field back into the custom type. This allows existing operations (like ExpressionFunction or RegexFilter) to operate on values owned by a custom type. For example, if you have a Person object, have a Function named GetPersonAge that takes Person as an argument and only returns the age as the result. The next operation can then Filter all Persons based on their age. This may seem like more work and less effiicient, but it keeps your application flexible and reduces the amount of duplicate code (the only alternative here is to create a PersonAgeFilter which results in one more thing to test).

8.12 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 );`

8.13 Look at the Source Code

When in doubt, look at the Cascading source code. If something is not documented in this User Guide or JavaDoc and its a feature of Cascading, the source code will give you clear instructions on what to do or expect.

9. CookBook

Some common idioms used in Cascading applications.

9.1 Tuples and Fields

Copy a Tuple instance

```
Tuple original = new Tuple( "john", "doe" );

// call copy constructor
Tuple copy = new Tuple( original );

assert copy.get( 0 ).equals( "john" );
```

Nest a Tuple instance within a Tuple

```
Tuple parent = new Tuple();
parent.add( new Tuple( "john", "doe" ) );

assert ( (Tuple) parent.get( 0 ) ).get( 0 ).equals( "john" );
```

9.2 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 Each( pipe, new Fields( "keepField" ), new Identity(),
  Fields.RESULTS );
// 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 arguments = new Fields( "longField", "booleanField" );
Class types[] = new Class[]{long.class, boolean.class};
Identity identity = new Identity( types );

// convert from string to given type, inline replace values
pipe = new Each( pipe, arguments, identity, Fields.REPLACE );
```

Insert constant values into a stream

```
Fields fields = new Fields( "constant1", "constant2" );
pipe = new Each( pipe, new Insert( fields, "value1", "value2" ),
  Fields.ALL );
```

9.3 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 );
```

9.4 Stream Ordering

Remove duplicate Tuples in a stream

```
// group on all values
pipe = new GroupBy( pipe, Fields.ALL );
// only take the first tuple in the grouping, ignore the rest
pipe = new Every( pipe, Fields.ALL, new First(), Fields.RESULTS );
```

Create a list of unique values

```
// group on all unique 'ip' values
pipe = new GroupBy( pipe, new Fields( "ip" ) );
// only take one 'ip' tuple in the group
pipe = new Every( pipe, new Fields( "ip" ), new First(),
  Fields.RESULTS );
```

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 );
```

9.5 API Usage

Pass properties to a custom Operation

```
// set property on Flow
Properties properties = new Properties();
properties.put( "key", "value" );
FlowConnector flowConnector = new FlowConnector( 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
Tap sourceLeft = new Hfs( new Fields( "some-fields" ), "some/path" );
Tap sourceRight = new Hfs( new Fields( "some-fields" ), "some/path" );

Pipe[] pipesArray = Pipe.pipes( headLeft, headRight );
Tap[] tapsArray = Tap.taps( sourceLeft, sourceRight );

// a convenience function for creating branch names to tap maps
Map<String, Tap> sources = Cascades.tapsMap( pipesArray, tapsArray );

// sink taps
Tap sinkLeft = new Hfs( new Fields( "some-fields" ), "some/path" );
Tap sinkRight = new Hfs( new Fields( "some-fields" ), "some/path" );

pipesArray = Pipe.pipes( tailLeft, tailRight );
tapsArray = Tap.taps( sinkLeft, sinkRight );

// or create the Map manually
Map<String, Tap> sinks = new HashMap<String,Tap>();
sinks.put( tailLeft.getName(), sinkLeft );
sinks.put( tailRight.getName(), sinkRight );

// set property on Flow
FlowConnector flowConnector = new FlowConnector();
```

```
Flow flow = flowConnector.connect( "flow-name", sources, sinks, tailLeft, tailRight )
```

10. How It Works

10.1 MapReduce Job Planner

The 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 under the scenes. As is the scheduling of the individual MapReduce jobs, and the clean up of intermediate data sets that bind the jobs together.
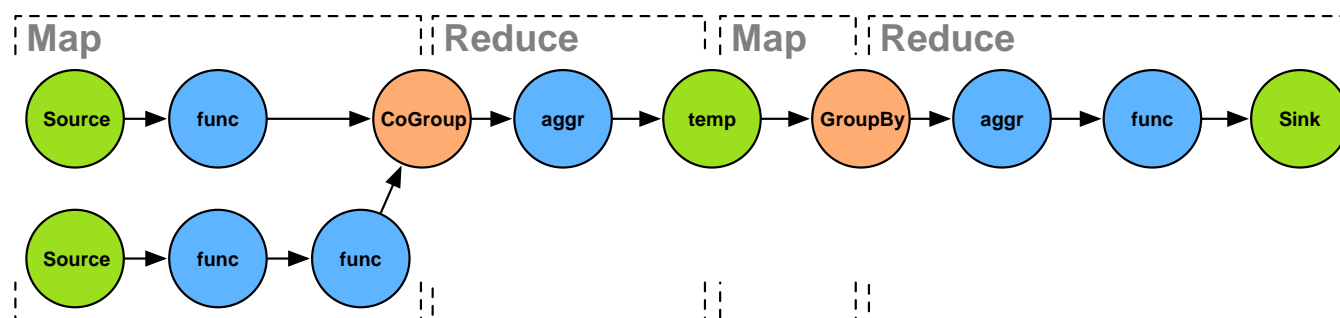
Above we can see how a reasonably normal Flow would be partitioned into MapReduce jobs. Every job is delimited by a temporary file that is the sink from the first job, and then the source to the next job.

To see how your Flows are partitioned, call the `Flow#writeDOT()` method. This will write a DOT [http://en.wikipedia.org/wiki/DOT_language] file out to the path specified, and can be imported into a graphics package like OmniGraffle or Graphviz.

10.2 The Cascade Topological Scheduler

Cascading has a simple class, `Cascade` , that will take a collection of Cascading Flows and execute them on the target cluster in dependency order.

Consider the following example.

- Flow 'first' reads input file A and outputs B.

- Flow 'second' expects input B and outputs C and D.

- Flow 'third' 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 it's incoming edges (files) available, it will be scheduled on the cluster.

In the example above, 'first' goes first, 'second' goes second, and 'third' is last.

If two or more Flows are independent of one another, they will be 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 behaviour a compiler would exhibit if a source file wasn't updated before a recompile.

This is very handy if you have a large number of jobs that should be executed as a logical unit with varying dependencies between them. Just pass them to the CascadeConnector, and let it sort them all out.