# Cascading guide

目录

[Cascading guide 1](#_Toc468179967)

[Introduction 1](#_Toc468179968)

[What Is Cascading? 1](#_Toc468179969)

[Another Perspective 1](#_Toc468179970)

[Why Use Cascading? 2](#_Toc468179971)

[The Cascading Philosophy 3](#_Toc468179972)

[Who Are the Users? 3](#_Toc468179973)

## Introduction

### What Is Cascading?

Cascading is a data processing API and processing planner used for defining, sharing, and executing data-oriented applications. These applications can execute on a single computing node or distributed computing platform with minimal code changes.

Cascading applications can be written in a way that allows the developer to create data-parallel applications that minimize the coupling to any specific computing platform. Fewer platform dependencies optimizes portability.

On a single node, Cascading’s "local mode" can be used to efficiently test code and process local files before being deployed on a cluster.

On a distributed computing cluster, Cascading natively supports the Apache Hadoop platform. Specific planners for both MapReduce and the Apache Tez directed acyclic graphs (DAGs) are provided.

Additionally, Cascading can support new platforms as they emerge so business logic will not need to be rewritten in order to leverage state-of-the-art technologies.

|  |  |
| --- | --- |
|  | Cascading is open-source under the Apache 2.0 License. |

### Another Perspective

Another way to look at Cascading is to consider the typical stack behind running a query with syntax like SQL:

1. The query syntax is parsed to an abstract syntax tree (AST).
2. The AST is logically reduced to an intermediate model. During this processing, many logical optimizations can be applied.
3. The intermediate model is translated into an executable model. Much of the processing between the intermediate and executable models applies physical optimizations.

Cascading provides both the intermediate model, and a planner that will translate that model into a physical executable model. The intermediate model is presented as a Java API, and the planner is both modifiable and pluggable.

### Why Use Cascading?

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

* Increasing data size
* Increasing computation requirements
* Increasing complexity in data centers
* Increasing need for accountability and manageability

Cascading applications scale to load seamlessly. The business logic for small data that may become big data does not require changes to the application logic.

Cascading provides many core data processing primitives and operations. Business logic can be coded directly against the Cascading APIs, or higher order frameworks or languages can be created on top. These frameworks or languages can be used to improve developer or analyst productivity for a specific use-case or type of enterprise.

Cascading as a platform allows for integration and business logic to remain decoupled until runtime so that different storage platforms can be bound to the logic on demand. New platforms can be leveraged during computation as business needs or infrastructure resources change.

Cascading can be managed. Besides a number of core features that allow for Cascading applications to be easily operationalized, Cascading works with commercial tools so that applications can be monitored in real-time. Developers gain in-depth understanding of application behavior at runtime in order to improve performance and reliability. Operations teams that are responsible for managing business processes can easily monitor application status.

### The Cascading Philosophy

Cascading was designed to be extensible, to behave deterministically, and to fail fast where possible.

Many default features and behaviors of Cascading can be replaced or overridden. Where the Cascading User Guide does not provide guidance, review the [Javadoc](http://cascading.org/documentation/) for APIs and properties that can be used or modified.

The execution plans created by the Cascading planner are both intuitive and stable:

* As processing logic is added during application development, the resulting plan is only an incremental increase in complexity and is proportional to the changes that were just applied. After becoming familiar with Cascading, you can easily understand the implication of adding a new CoGroup to the assembly and how it will behave on the cluster.
* Every execution plan is the same when there are no code changes. Across versions of Cascading, any likely behavioral differences are documented (if not accompanied by a way to revert the behavior).
* The Cascading planner creates all units of work up front. The planner verifies all dependencies are met from available resources like sources and sinks, down to the field level. If a downstream operation requires a specific field (for example, "zipcode"), the planner guarantees it is available upstream from a data source or operation.

Many of these ideals seem obvious, but many systems regenerate plans on the fly or attempt shortcuts during execution. Even if these methods of other systems are successful, there is no certainty that they result in improved runtime performance.

It is not acceptable to fail part way, in a non-deterministic (non-repeatable) fashion, for applications that may run hours or days at a stretch.

### Who Are the Users?

Cascading users typically fall into three roles:

The **application executor** is a person (for example, a developer or analyst) or process (for example, a cron job) that runs a data processing application on a given cluster. This is typically done via the command line, using a prepackaged JAR file compiled against the Cascading libraries. The application can accept command-line parameters for customization during execution.

The **process assembler** assembles data processing workflows into unique applications. This work is generally a development task that involves chaining together operations to act on input data sets so that they produce output data sets. Development can be done with the default Java Cascading API, the [Fluid](http://cascading.org/fluid/) API, or with a scripting language such as Scala, Clojure, Groovy, JRuby, Jython. Cascading also supports domain-specific languages that are implemented in the scripting languages.

The **operation developer** writes individual functions or operations (typically in Java) or reusable subassemblies that act on the data that passes through the data processing workflow. A trivial 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. Operations can execute at any granularity, from simply parsing a string to performing complex procedures on the argument data using third-party libraries. Cascading provides many prebuilt operations.

Each of the three roles can be filled by a developer. But in some organizations non-developers might run ad-hoc applications or build production processes because Cascading supports clean separation of the responsibilities that are entailed in the three roles.

As of Cascading 3.0, two new roles have emerged:

The **platform optimizer** improves execution runtime or resource utilization. This role can be responsible for creating workload-specific query-plan rules that target specific improvements. Another responsibility of the platform optimizer could be to add new core primitives that can be leveraged in application logic.

The **platform developer** ports Cascading to new computing platforms. This is an emerging API, but already proven to be very robust and powerful. As business needs change and new technologies emerge, a developer can create bindings to these new technologies allowing existing investments in the Cascading API and broader ecosystem to be leveraged.

## Diving into the APIs

### Anatomy of a Word-Count Application

The most common example presented to new developers is an application that counts words. It is the data processing equivalent to a "Hello World" application.

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

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

Example 1. Word Counting

// define source and sink Taps.

Scheme sourceScheme = new TextLine( new Fields( "line" ) );

Tap source = new Hfs( sourceScheme, inputPath );

// 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 );

Scheme sinkScheme = new TextDelimited( new Fields( "word", "count" ) );

Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );

// initialize app properties, tell Hadoop which jar file to use

Properties properties = AppProps.appProps()

.setName( "word-count-application" )

.setJarClass( Main.class )

.buildProperties();

// plan a new Flow from the assembly using the source and sink Taps

// with the above properties

FlowConnector flowConnector = new Hadoop2MR1FlowConnector( properties );

Flow flow = flowConnector.connect( "word-count", source, sink, assembly );

// execute the flow, block until complete

flow.complete();

|  |  |
| --- | --- |
|  | Read each line of text from a file and give it the field name "line" |
|  | Parse each "line" into words with the RegexGenerator object, which returns each word in the field named "word" |
|  | Sort and group all the tuples on the "word" field, using the GroupBy object |
|  | Count the number of elements in each group, using the Count object, and store this value in the "count" field |
|  | Write out the "word" and "count" fields |
|  | Set application-specific metadata to allow the application to run |
|  | Choose the platform for execution |
|  | Plan the unit of work (a Flow) to be executed |
|  | Start the flow and wait until it is completed |

Several features of this example are worth highlighting:

* The pipe assembly is not coupled to the data (i.e., the Tap instances) until the last moment before execution. File paths or references are not embedded in the pipe assembly. Instead, the pipe assembly is specified independent of data inputs and outputs. The only dependency is the data scheme (i.e., the field names).
* In Cascading, every input or output file has field names associated with its contents, and every processing element of the pipe assembly either expects the specified fields from upstream or creates them. This allows developers to easily self-document their code. The Cascading planner "fails fast" if an expected dependency between elements is not satisfied — for instance, if a needed field name is missing or incorrect.
* Pipe assemblies are assembled through constructor chaining. This may seem odd, but it is done for two reasons. First, constructor chaining allows each object to be immutable. Second, it prevents developers from creating "cycles" (i.e., recursive loops) in the resulting pipe assembly. Recursive loops hinder the generation of insightful directed acyclic graphs (DAGs) from pipe assemblies. (If looping processes are desired, there are safer approaches to achieving this result.)
* The very first Pipe instance has a name. That instance is the head of this particular pipe assembly. Pipe assemblies can have any number of heads, and any number of tails. Although the tail in this example does not have an explicit name, in a more complex assembly it would (since this assembly is a single branch, the tail inherits the head name).

Heads and tails of pipe assemblies are assigned names to disambiguate them. One reason is that names are used to bind sources and sinks to pipes during planning. (The example above is an exception, because there is only one head and one tail — and consequently only one source and one sink—​so the binding is unmistakable.) Another reason is that the naming of pipes contributes to self-documentation of pipe assemblies, especially where there are splits, joins, and merges in the assembly.

### Fluid: An Alternative Fluent API

The above example is using the default "raw" Cascading API. This API is handy when creating very complex business logic, or creating frameworks that may need to algorithmically generate Cascading assemblies. [Lingual](http://www.cascading.org/lingual/), an ANSI SQL layer over Cascading, leverages this API.

An alternative API is called [Fluid](http://www.cascading.org/fluid/). Currently Fluid is maintained in a different project, but there are plans to make it available as part of all future Cascading releases. The goal is to provide a build plugin that generates fluent APIs from any custom code that is run in the build.

Example 2 shows how the same word-count application that appeared in Example 1 could be coded with the Fluid fluent API (with all comments removed).

Example 2. Word Counting with Fluid

Scheme sourceScheme = new TextLine( new Fields( "line" ) );

Tap source = new Hfs( sourceScheme, inputPath );

Pipe assembly = Fluid.assembly()

.startBranch( "wordcount" )

.each( new Fields( "line" ) )

.function(

Fluid.function()

.RegexGenerator()

.fieldDeclaration( new Fields( "word" ) )

.patternString( "(?<!\\pL)(?=\\pL)[^ ]\*(?<=\\pL)(?!\\pL)" ).end()

)

.outgoing( Fields.RESULTS )

.groupBy( new Fields( "word" ) )

.every( Fields.ALL )

.aggregator(

Fluid.aggregator().Count( new Fields( "count" ) )

)

.outgoing( Fields.ALL )

.completeGroupBy()

.completeBranch();

Scheme sinkScheme = new TextDelimited( new Fields( "word", "count" ) );

Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );

Properties properties = AppProps.appProps()

.setName( "word-count-application" )

.setJarClass( Main.class )

.buildProperties();

FlowConnector flowConnector = new Hadoop2MR1FlowConnector( properties );

Flow flow = flowConnector.connect( "word-count", source, sink, assembly );

flow.complete();

|  |  |
| --- | --- |
|  | Read each line of text from a file and give it the field name "line" |
|  | Parse each "line" into words with the RegexGenerator object, which returns each word in the field named "word" |
|  | Sort and group all the tuples on the "word" field, using the GroupBy object |
|  | Count the number of elements in each group, using the Count object, and store this value in the "count" field |
|  | Write out the "word" and "count" fields |
|  | Set application-specific metadata to allow the application to run |
|  | Choose the platform for execution |
|  | Plan the unit of work (a Flow) to be executed |
|  | Start the flow and wait until it is completed |

The remainder of this Cascading User Guide focuses on the default Cascading Java API introduced in Example 1.

http://www.cascading.org/documentation/tutorials/