Jumbo

User guide

# Introduction

If you’ve gone through the quick start guide, by this point you will have installed Jumbo and run some jobs using the provided samples. In this document we will look at how to create your own data processing jobs. First, a sample will be used to introduce the basic programming concepts. Then, the features of Jumbo Jet are explained in more detail. Finally, a more advanced sample is given that demonstrates some of these features.

A data processing job in Jumbo Jet consists of a linear sequence of *stages*, connected by *channels*. A stage performs some processing operation on the data, and the channels control how data is transferred between stages. Stages are responsible for the bulk of the data processing work, while channels transfer data and can do some additional intermediate processing such as sorting and partitioning.

If you are familiar with MapReduce, you can consider MapReduce to be a two-stage job where the first stage runs the map function, the second runs the reduce function, and the channel between the two stages sorts and partitions the data. In fact, that is exactly how you create MapReduce jobs in Jumbo. But Jumbo is not limited to MapReduce; it can use any kind of job structure.

A stage reads data either from a *data input* (typically a file or multiple files on the DFS) or a channel, and writes data either to a channel or a *data output* (typically a directory on the DFS). A stage can also have multiple channels as input, for example to perform a join.

Stages are divided into tasks, where each task processes a part of the data. Tasks are run in parallel on multiple systems in a cluster. For stages reading a data input (file), the input is split linearly to divide it across tasks. For stages reading a channel input, the channel partitions the data across the tasks.

Tasks run a user-defined piece of code to do their processing. This code doesn’t need to be aware of most of the details. Regardless of whether the task is reading from or writing to a file or a channel, the code is the same. Input is provided via a RecordReader and output is written to a RecordWriter, which take care of the details. Since partitioning (and optionally, things like sorting) are handled by the Jumbo Jet infrastructure, you don’t have to worry about how to perform those operations; you simply need to specify you want them to happen.

The typical way to create a job in Jumbo is to use the JobBuilder, which allows you to specify a sequence of operations which are then translated into a job configuration of stages and channels. This means you can create jobs without worrying too much about their actual structure during execution (although of course this is available if you want to do more complex processing).

# Tutorial: creating a data processing job using the JobBuilder

In order to introduce data processing with Jumbo, we are going to look how to create the WordCount sample job. WordCount reads an input file and counts how often each word occurs in the input. It’s a pretty standard example for MapReduce style data processing.

If you’re using Visual Studio to do this tutorial, you should create a new C# class library project (note: you can use any Mono or .Net language, but this example is in C#), and add references to Ookii.Jumbo.dll, Ookii.Jumbo.Dfs.dll, Ookii.Jumbo.Jet.dll, and Ookii.CommandLine.dll.

If not using Visual Studio, create a new C# file called WordCount.cs. You can compile this file to a DLL from the command like with csc for .Net or mcs for Mono, making sure to reference the appropriate libraries.

## Creating a JobRunner

Although you can write your own client application that submits jobs to Jumbo, the easiest way is to create a JobRunner.

A JobRunner is a class that creates the job configuration, and specify command line arguments for the job’s invocation. These JobRunners are invoked using the JetShell job command that we used in the quick start guide.

A JobRunner is any class that implements the IJobRunner interface, although typically you’ll probably want to inherit from the BaseJobRunner class, which defines a number of standard command line arguments and behaviors for JobRunners. Because we’re going to use the JobBuilder to build our job, we’ll use the JobBuilderJob class as the base class for our job runner.

So, start out a new C# file as follows:

$lang: C#; WordCount JobBuilderJob$  
using System;  
using Ookii.CommandLine;  
using Ookii.Jumbo.IO;  
using Ookii.Jumbo.Jet.Jobs.Builder;  
  
namespace JumboSample  
{  
 public class WordCount : JobBuilderJob  
 {  
 }  
}

That’s our job builder class. The first thing we need to add is the arguments. This job is going to need to know the input and output path, so we’ll have to create command line arguments for that. Jumbo uses [Ookii.CommandLine](http://ookiicommandline.codeplex.com) for defining command line arguments, so we need to add properties and mark them as command line arguments:

$lang: C#; CommandLineArgument$  
[CommandLineArgument(IsRequired = true, Position = 0)]  
public string InputPath { get; set; }  
  
[CommandLineArgument(IsRequired = true, Position = 1)]  
public string OutputPath { get; set; }

This creates two required positional arguments, so it is now possible to specify the input and output path on the command line when invoking this job.

## Data processing functions

In order to create a job, we need to write functions that process the data for each of the stages. WordCount is a distributed counting operation, which consists of two operations: the first operation extracts the tokens (words) and initializes their count to 1. The second step aggregates the counts for each token (word).

For this job, we’re going to read the input line-by-line, so the function for the first operation should split that line into words, and then generate key/value pairs with the word as the key and a count of 1 as the value. For that purpose, we’ll write a map function, much like you’ll see in Hadoop MapReduce jobs (only here it’s just a function; unlike in Hadoop there is no need to create full classes for each task if you’re using the JobBuilder):

$lang: C#; Utf8String RecordWriter Pair Utf8String StringSplitOptions$  
public static void MapWords(Utf8String line, RecordWriter<Pair<Utf8String, int>> output)  
{  
 string[] words = line.ToString().Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);  
 foreach( string word in words )  
 output.WriteRecord(Pair.MakePair(new Utf8String(word), 1));  
}

First of all, notice that this is a public static method. This is preferred for processing functions. Although you can use private methods or even lambdas, these must be called through a delegate which is slower than the direct call that’s possible with public static methods. Keep in mind that this code will be executed in an entirely different process than the one that is building the job, so none of the state that’s available during job creation will still be intact during execution. Preferably, processing functions should not use any external state.

Let’s see how this function works. The function will be called for each record in the input, which in this case are the lines of text in the input (unlike Hadoop, records don’t have to be key/value pairs, so the input record in this case is not). That line is passed in the first parameter, line.

The type of this parameter is Utf8String. This is a special string type used by Jumbo; unlike the regular String class, it is mutable and also uses a more compact in-memory representation for most strings (as the name suggests, text is stored in utf-8 encoding). These two features make Utf8String more efficient for Jumbo’s purposes. Although Jumbo can use regular strings, it’s recommended to use Utf8String unless you have a good reason not to. In this case, we’ll be reading the input using the LineRecordReader, which returns Utf8String records, so we have to use it for the input.

The second parameter is the RecordWriter to which the output should be written. This record writer can be connected to a channel, a file, and can use a host of different output serialization options. The processing function doesn’t need to care; it just writes records to the writer.

The type of our output records is Pair<Utf8String, int>. Note that you *cannot* use the regular KeyValuePair structure with Jumbo; Pair provides a number of additional features that Jumbo needs.

The function body simply splits the input on spaces, and creates a key/value pair for each word with the key as the word, and a value of 1.

For the second step, we’re going to use Jumbo’s support for hash table aggregation (rather than a typical reduce task). This means we need to define a function that takes the current value of a key, a new value, and returns the updated value. In this case, we need to sum the values:

$lang: C#; Utf8String$  
public static int AggregateCounts(Utf8String key, int oldValue, int newValue)  
{  
 return oldValue + newValue;  
}

Note that we’re not using the key parameter; this function just adds the values, and doesn’t need the key. However, this is the signature that the JobBuilder requires, so the parameter must be included.

In this case, it’s not necessary to write output. Jumbo has a built-in task type (AccumulatorTask) that performs aggregation which takes care of all of that. This function is called by that task to update the values, so that’s all the code we need to write for aggregation functions.

## Creating the job

Now that we have the functions that will process the data, we need to tell Jumbo which order to apply them in. This is where the JobBuilder comes in. It provides a number of methods to construct a sequence of various types of data processing operations. The resulting job configuration can be customized to specify things like the number of partitions or channel types, or you can just use JobBuilder’s defaults, which automatically decide on a number of partitions based on the input data size and task capacity of your cluster.

In order to create the job with a JobRunner derived from JobBuilderJob, you need to override the BuildJob function. Here’s that function for our word count sample:

$lang: C#; JobBuilder Utf8String Pair LineRecordReader TextRecordWriter$  
protected override void BuildJob(JobBuilder job)  
{  
 var input = job.Read(InputPath, typeof(LineRecordReader));  
 var words = job.Map<Utf8String, Pair<Utf8String, int>>(input, MapWords);  
 var aggregated = job.GroupAggregate<Utf8String, int>(words, AggregateCounts);  
 WriteOutput(aggregated, OutputPath, typeof(TextRecordWriter<>));  
}

Let’s see what this function does. The first thing to remember is that none of the operations are actually executed here. Each of these functions doesn’t actually perform the specified operation, but adds a stage to the job configuration that will perform that operation. Only when the job is submitted to the cluster will this be executed.

The first line tells Jumbo to read input from the specified path (which we get from the command line argument) using the LineRecordReader. This record reader reads utf-8 text input and provides a Utf8String record for each line. Since we didn’t specify any options, Jumbo will split the input file based on the DFS block size for the file (if the input is being read from the Jumbo DFS), and create a single task for each block. The LineRecordReader handles those splits and makes sure that all records are read by exactly one task (no records are missed and none are read by two tasks), even when those records cross a block boundary. Your code doesn’t need to worry about that.

It is possible to use the properties on the input variable to customize the split sizes to create more or fewer tasks, but in this simple example, we just use the defaults.

The second line of the BuildJob function tells Jumbo to invoke a map function (the MapWords function we defined earlier) for each record in the input. Unfortunately, limitations in C#’s type argument inference when it comes to delegates means that it’s necessary to explicitly specify the generic arguments for the JobBuilder.Map function. This is true for all JobBuilder functions that use delegates.

The Map function creates a stage that reads from the input, and executes the specified function on each record. What to do with the output is specified later. As indicated before, Jumbo will automatically create multiple tasks in the stage based on the input split size.

The third line calls the GroupAggregate function, which tells Jumbo to group data by key and run the specified aggregation function (the AggregateCounts function we wrote earlier). Because the input here is the output from another operation, the JobBuilder will create a channel and set defaults for the number of partitions, which you can of course override if you want.

GroupAggregate actually creates two stages, both performing the aggregation operation. The first does it locally for each task in the input stage, and the second aggregates the data from all tasks. This is similar to using a combiner with MapReduce in Hadoop, and helps reduce the amount of data that needs to be transferred over the network.

The final line tells Jumbo to write the output to the specified path using the specified RecordWriter (in this case, we’re writing the output as text). Note that we’re not calling the JobBuilder.Write method directly, but instead use the JobBuilderJob.WriteOutput method. This method applies some settings from command line arguments that are common to each JobBuilderJob to the output. These command line arguments allow the user to specify things like the block size and replication factor for the output.

And that’s it. Putting it all together gives us this:

$lang: C#; WordCount JobBuilderJob CommandLineArgument JobBuilder LineRecordReader Utf8String Pair TextRecordWriter StringSplitOptions$  
using System;  
using Ookii.CommandLine;  
using Ookii.Jumbo.IO;  
using Ookii.Jumbo.Jet.Jobs.Builder;  
  
namespace JumboSample  
{  
 public class WordCount : JobBuilderJob  
 {  
 [CommandLineArgument(IsRequired = true, Position = 0)]  
 public string InputPath { get; set; }  
  
 [CommandLineArgument(IsRequired = true, Position = 1)]  
 public string OutputPath { get; set; }  
  
 protected override void BuildJob(JobBuilder job)  
 {  
 var input = job.Read(InputPath, typeof(LineRecordReader));  
 var words = job.Map<Utf8String, Pair<Utf8String, int>>(input, MapWords);  
 var aggregated = job.GroupAggregate<Utf8String, int>(words, AggregateCounts);  
 WriteOutput(aggregated, OutputPath, typeof(TextRecordWriter<>));  
 }  
  
 public static void MapWords(Utf8String line, RecordWriter<Pair<Utf8String, int>> output)  
 {  
 string[] words = line.ToString().Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);  
 foreach( string word in words )  
 output.WriteRecord(Pair.MakePair(new Utf8String(word), 1));  
 }  
  
 public static int AggregateCounts(Utf8String key, int oldValue, int newValue)  
 {  
 return oldValue + newValue;  
 }  
 }  
}

# MapReduce jobs

As you have seen, Jumbo supports a number of features that take it beyond just MapReduce, like the hash table aggregation used in the word count sample, or the ability to have jobs with more than two stages.

It is however possible to create a normal MapReduce job. Essentially, all that’s needed is two-stage job where the first stage runs a map function, the second stage runs a reduce function, and the channel between them sorts the data by key. The JobBuilder provides methods for all these operations.

For example, to convert the word count sample to MapReduce, all we need to do is replace the AggregateCounts function with a reduce function:

$lang: C#; Utf8String IEnumerable RecordWriter Pair  
public static void ReduceWordCount(Utf8String key, IEnumerable<int> values, RecordWriter<Pair<Utf8String, int>> output)  
{  
 output.WriteRecord(Pair.MakePair(key, values.Sum()));  
}

This is pretty much exactly like the reduce function you’d write in Hadoop (except for convenience, I used the Sum method provided by LINQ in .Net, rather than summing the values manually as you’d have to do in Java).

The BuildJob function for the MapReduce version would look as follows:

$lang: C#; LineRecordReader Utf8String Pair TextRecordWriter  
var input = job.Read(InputPath, typeof(LineRecordReader));  
var words = job.Map<Utf8String, Pair<Utf8String, int>>(input, MapWords);  
var sorted = job.SpillSortCombine<Utf8String, int>(words, ReduceWordCount);  
var counted = job.Reduce<Utf8String, int, Pair<Utf8String, int>>(sorted, ReduceWordCount);  
WriteOutput(counted, OutputPath, typeof(TextRecordWriter<>));

It starts off the same as the previous version: reads the input, and runs the map function. It then calls SpillSortCombine, which performs an external merge sort using multiple passes on the sending stage’s side, and merging the data on the receiving stage’s side. This is identical to the sorting method used by Hadoop 1.0, and can handle very large amounts of data without putting too much pressure on memory usage. Like Hadoop, it’s possible to run a combiner during the sort, for which in this case we use the ReduceWordCount function.

After sorting, we call the Reduce function to add a stage that runs the ReduceWordCount function, and finally we write the output as in the previous sample.

The end result is a job that runs almost exactly like a Hadoop job would.

# Jumbo features

This section provides more detail about the major features of Jumbo Jet.

## Job Configuration

The configuration for a job is specified as an XML file, which indicates for every stage what its inputs and outputs are, what its task type is, and other options such as the partitioner or multi-input record reader for each channel. You can also add settings to the configuration which can be accessed by your tasks.

When you work with the JobBuilder, it constructs a sequence of operations from the methods you call which is then translated into a configuration. It is possible to manually configure a job using the JobConfiguration class, or if you really want you could manually write the XML file. However, usually using the JobBuilder is the best option. If you’re using a job runner derived from JobBuilderJob, you can override the OnJobCreated method to customize elements of the configuration after the job configuration has been created.

## Tasks

A task executes a data processing operation over a portion of the stage’s input data. The operation to execute is implemented by a class that implements the ITask<TInput, TOutput> generic interface. The Run method receives a RecordReader that it can use to read records from the input, and a RecordWriter to write records to the output.

A task class can implement IConfigurable (or inherit from Configurable to use a default implementation) to access various aspects of the job and task configuration during execution.

Jumbo provides several built-in task types. One important task type is PushTask<TInput, TOutput>, which defines a method that will be called for each record (similar to a map function). If the receiving stage of a pipeline channel (see below) inherits from PushTask, the RecordWriter for the channel will directly call the ProcessRecord function for each record, bypassing the Run method. This provides better performance for pipeline channels.

The other built-in task types are utility task types for things like aggregation or reduce tasks, and are not treated specially.

When using the JobBuilder methods that take a delegate, the JobBuilder generates a task type that calls the target method of the delegate (directly if the method is public and static, or through the delegate if not).

## Record Reuse

In order to reduce the number of objects created to alleviate the pressure on the Garbage Collector and improve performance, it is desirable to be able to reuse the same object instance for every record a task. For this reason, most RecordReaders that can be used with Jumbo Jet are able to reuse the same object instance every time you read a record.

However, Jumbo doesn’t know automatically when that is safe to do. It doesn’t know if the task type stores record object instances in memory or something similar that may make record reuse unsafe. Because of this, a task must declare that it can handle reused object instances for the input by applying the AllowRecordReuseAttribute to the task type (with the JobBuilder methods that use delegates, you can apply this attribute to the delegate’s target method, or use an overload that takes a RecordReuseMode). In general, you should always apply this attribute unless you have a reason not to do so.

Output records can also be reused. All record writers that can be used with Jumbo Jet are required to support receiving the same object instance for each record. The record writers for the channels also satisfy this criteria. The only situation where you need to take care is with pipeline channels, where Jumbo will directly pass the output record objects to the receiving task of the pipeline channel. Therefore, reusing output records is only safe if the receiving stage of the pipeline channel allows record reuse. If you’re writing a task for use in a single job, you’ll probably know when that is safe or not. If you’re writing a reusable task type, check the TaskContext.StageConfiguration.AllowOutputRecordReuse property and alter your task’s behavior accordingly.

If your task passes record object instances from the input directly to the output, you should set the PassThrough property of the AllowRecordReuseAttribute to true, in which case Jumbo will only reuse input records if the output of the stage that uses the task also allows record reuse.

Record reuse is not an issue if the record type is a value type (a basic type or a struct).

## Channels

Channels determine how data is transferred between the tasks of different stages. Channels are responsible for transferring the data, but also for partitioning output between different tasks, and merging input from different tasks.

The stage that writes to a channel is called the *sending stage*, and the stage that reads from it the *receiving stage*.

Jumbo has three types of channel, file, TCP, and pipeline:

* **File channels** store intermediate data in a file on the local disk of each task of the sending stage, and the tasks in the receiving stage shuffle these files across the network. This behavior is similar to how the shuffle stage is Hadoop 1.0 works, and offers the best fault tolerance. File channels should be used in most cases.
* **TCP channels** make a direct connection between tasks to transfer data directly without materializing it. TCP channels put limitations on the job structure, because it is required that all tasks in the receiving stage can be executed simultaneously. TCP channels also disable task level fault tolerance. Because I didn’t use this feature often, they are also not as well tested as file channels.
* **Pipeline channels** (also referred to as in-process channels) are used to combine two tasks into one. Unlike file and TCP channels, it therefore connects only two tasks, rather than reading data from all tasks in the sending stage. Two tasks connected by pipeline channel run on the same node, in the same process (and are treated by the scheduler as a single task), so object instances are directly passed between the two tasks. The receiving stage of a pipeline channel is also called a *child stage* of the sending stage.

You can specify a partitioner for a channel that is responsible for dividing the data over the receiving stage’s tasks. The default partitioner hashes the records (by key if it’s a key/value pair). You can implement custom partitioners by creating a class that implements IPartitioner<T>.

You can also specify a multi-input record reader that is responsible for merging the data from all the tasks in the sending stage (more about this below).

## Record types

The basic .Net Framework types SByte, Byte, UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal, DateTime and String can all be used as record types in Jumbo. In addition, Jumbo provide the Utf8String, Pair<TKey, TValue>, and WritableCollection<T> record types.

In order to use a custom type as a record, that type must either implement IWritable or provide a ValueWriter. Typically, you use IWritable for reference types (classes), and ValueWriter for value types (structs). You *cannot* use IWritable for value types.

In certain circumstances, additional interfaces may be required by tasks or channels. In general, it is good practice to implement IComparable<T>, IEquatable<T>, and (for reference types only) ICloneable on all record types. You should also implement the Object.Equals, Object.GetHashCode and Object.ToString methods.

### Implementing IWritable

The IWritable interface provides two methods, Read and Write.

The Read method must read the object’s entire state from a BinaryReader. Jumbo constructs object instances using System.Runtime.Serialization.FormatterServices.GetUninitializedObject, so the constructor will not be called. Therefore, Read may not assume any prior state and must place the object in a valid state. Because the same object instance may be reused for multiple records, the Read method may not leave any state from the previous record intact, but must replace the entire state for the next record. A call to Read must read exactly all data written by the Write data of the same object, and no more.

The Write method must serialize the objects entire state to a BinaryWriter. All state belonging to the record must be serialized, and a call to the Read method on the same type using the data written must reconstruct the state of the object that Write was called on.

### Creating a ValueWriter

IWritable cannot be used with immutable types or value types. In those case, create a separate type that implements IValueWriter<T>, whose Read and Write methods follow the same rules as those of IWritable. Apply the ValueWriterAttribute to the record type to indicate the type of its ValueWriter.

## Record readers and writers

Record readers and writers control the file format of the input and output of a job. Jumbo provides the built-in LineRecordReader and TextRecordWriter<T> to read and write text data, and the RecordFileReader<T> and RecordFileWriter<T> to read and write data from a structured format similar to sequence files in Hadoop. The BinaryRecordReader<T> and BinaryRecordWriter<T> are used for intermediate data; it’s not recommended to use these as input or output formats.

### Custom record writers

A custom record writer must inherit from the RecordWriter<T> class (it is not sufficient to implement IRecordWriter, which is only provided for code that must handle a record writer without knowing the record type). If you write records to a stream, it’s recommended to inherit from StreamRecordWriter<T>.

A record writer may use any formatting it wants. There is no requirement to use IWritable, or that the format can be read back. It is not necessary for a record writer to have an accompanying record reader.

In order to use a record writer for output data in a Jumbo Jet job, it must satisfy the following criteria:

1. It must have a public constructor that takes a single parameter of type System.IO.Stream, and write records to that stream.
2. It must support record reuse, which means it can handle it if every call to WriteRecord uses the same object instance.

### Custom record readers

A custom record reader must inherit from the RecordReader<T> class (it is not sufficient to implement IRecordReader, which is only provided for code that must handle a record reader without knowing the record type). If you read records from a stream, it’s recommended to inherit from StreamRecordReader<T>.

A record reader may use any formatting it wants. There is no requirement to use IWritable, or that there is a corresponding record writer.

In order to use a record reader for input data in a Jumbo Jet job, it must satisfy the following criteria:

1. It must have a public constructor taking the following arguments:   
   *inputStream:* A System.IO.Stream to read the records from.  
   *offset:* A System.Int64 (long) that is the offset at which to start reading from the stream.  
   *size*: A System.Int64 (long) that is the number of bytes to read from the stream.  
   *allowRecordReuse:* A System.Boolean that indicates whether or not record reuse is allowed.
2. On the first call of ReadRecord, it must seek to *offset* and then find the start of the first record after that position, and start reading from there.
3. It must continue reading until the start of the next record is after *offset* + *size*, or until the end of the stream is reached. After this, ReadRecord must return false.
4. Given a particular *offset* and *size*, and two record readers a = new TheRecordReader(stream, offset, size, allowRecordReuse) and b = new TheRecordReader(stream, offset + size, size, allowRecordReuse), all records between *offset* and *offset* + *size* \* 2 must be read by either a or b, and there must be no record read by both a and b.
5. It should minimize the amount of data read beyond *offset* + *size*, because that data may be in another DFS block that may not be local.
6. It should behave correctly if the stream is a record stream that uses RecordStreamOptions.DoNotCrossBoundary.
7. If and only if *allowRecordReuse* is true, it is allowed to use the same object instance for every record. If *allowRecordReuse* is false, it must create a new instance to assign to CurrentRecord on every call to ReadRecord. Note that it is not required to reuse object instances if *allowRecordReuse* is true, although it is strongly recommended to support this.

Note that these requirements (particularly number 2) mean that a record reader that cannot find the start of the next record from an arbitrary stream offset cannot be used for input for a Jumbo Jet job. For example, BinaryRecordReader<T> does not meet this requirement.

### Multi-input record readers

A multi-input record reader is a record reader that combines input from multiple record readers. They are used by channels to combine the input of all the tasks in the sending stage, and are also used to combine the input from multiple channels in the case of a join.

Jumbo provides the following multi-input record readers for use with channels: MultiInputRecordReader<T> reads the input segments in the order they arrive, RoundRobinRecordReader<T> reads input records from all currently available segments in round-robin order, skipping segments that are waiting on input (this record reader is used for TCP channels), and MergeRecordReader<T> merges already sorted input segments into a completely sorted sequence (the JobBuilder.SpillSort method automatically uses this multi-input record reader type). The only multi-input record reader provided for reading input from multiple channels is the InnerJoinRecordReader<TOuter, TInner, TResult> which performs an inner join between exactly two stages.

To create a custom multi-input record reader, inherit from the MultiInputRecordReader<T> class (it is not sufficient to just implement IMultiInputRecordReader). Jumbo will call AddInput for every new input that becomes available. The order of inputs is not guaranteed. The input of a multi-input record reader may contain multiple partitions, which must be processed one by one. Additional partitions may be added after the multi-input record reader was created with the AssignAdditionalPartitions method.

For a channel multi-input record reader, the type of the records for all inputs will be the same as the record type of the multi-input record reader. For a stage multi-input record reader (which receives data from multiple channels), the input record types may be different, and the multi-input record reader should use the InputTypeAttribute to indicate which types it accepts.