| **Transformation** | **Description** |
| --- | --- |
| **Map** DataStream → DataStream | Takes one element and produces one element. A map function that doubles the values of the input stream:  DataStream**<**Integer**>** dataStream **=** *//...*  dataStream**.**map**(new** MapFunction**<**Integer**,** Integer**>()** **{**  @Override  **public** Integer **map(**Integer value**)** **throws** Exception **{**  **return** 2 **\*** value**;**  **}**  **});** |
| **FlatMap** DataStream → DataStream | Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words:  dataStream**.**flatMap**(new** FlatMapFunction**<**String**,** String**>()** **{**  @Override  **public** **void** **flatMap(**String value**,** Collector**<**String**>** out**)**  **throws** Exception **{**  **for(**String word: value**.**split**(**" "**)){**  out**.**collect**(**word**);**  **}**  **}**  **});** |
| **Filter** DataStream → DataStream | Evaluates a boolean function for each element and retains those for which the function returns true. A filter that filters out zero values:  dataStream**.**filter**(new** FilterFunction**<**Integer**>()** **{**  @Override  **public** **boolean** **filter(**Integer value**)** **throws** Exception **{**  **return** value **!=** 0**;**  **}**  **});** |
| **KeyBy** DataStream → KeyedStream | Logically partitions a stream into disjoint partitions, each partition containing elements of the same key. Internally, this is implemented with hash partitioning. See [keys](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#specifying-keys) on how to specify keys. This transformation returns a KeyedDataStream.  dataStream**.**keyBy**(**"someKey"**)** *// Key by field "someKey"*  dataStream**.**keyBy**(**0**)** *// Key by the first element of a Tuple* |
| **Reduce** KeyedStream → DataStream | A "rolling" reduce on a keyed data stream. Combines the current element with the last reduced value and emits the new value.   A reduce function that creates a stream of partial sums:  keyedStream**.**reduce**(new** ReduceFunction**<**Integer**>()** **{**  @Override  **public** Integer **reduce(**Integer value1**,** Integer value2**)**  **throws** Exception **{**  **return** value1 **+** value2**;**  **}**  **});** |
| **Fold** KeyedStream → DataStream | A "rolling" fold on a keyed data stream with an initial value. Combines the current element with the last folded value and emits the new value.   A fold function that, when applied on the sequence (1,2,3,4,5), emits the sequence "start-1", "start-1-2", "start-1-2-3", ...  DataStream**<**String**>** result **=**  keyedStream**.**fold**(**"start"**,** **new** FoldFunction**<**Integer**,** String**>()** **{**  @Override  **public** String **fold(**String current**,** Integer value**)** **{**  **return** current **+** "-" **+** value**;**  **}**  **});** |
| **Aggregations** KeyedStream → DataStream | Rolling aggregations on a keyed data stream. The difference between min and minBy is that min returns the minimum value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).  keyedStream**.**sum**(**0**);**  keyedStream**.**sum**(**"key"**);**  keyedStream**.**min**(**0**);**  keyedStream**.**min**(**"key"**);**  keyedStream**.**max**(**0**);**  keyedStream**.**max**(**"key"**);**  keyedStream**.**minBy**(**0**);**  keyedStream**.**minBy**(**"key"**);**  keyedStream**.**maxBy**(**0**);**  keyedStream**.**maxBy**(**"key"**);** |
| **Window** KeyedStream → WindowedStream | Windows can be defined on already partitioned KeyedStreams. Windows group the data in each key according to some characteristic (e.g., the data that arrived within the last 5 seconds). See[windows](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html) for a complete description of windows.  dataStream**.**keyBy**(**0**).**window**(**TumblingEventTimeWindows**.**of**(**Time**.**seconds**(**5**)));** *// Last 5 seconds of data* |
| **WindowAll** DataStream → AllWindowedStream | Windows can be defined on regular DataStreams. Windows group all the stream events according to some characteristic (e.g., the data that arrived within the last 5 seconds). See [windows](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html) for a complete description of windows.  **WARNING:** This is in many cases a **non-parallel** transformation. All records will be gathered in one task for the windowAll operator.  dataStream**.**windowAll**(**TumblingEventTimeWindows**.**of**(**Time**.**seconds**(**5**)));** *// Last 5 seconds of data* |
| **Window Apply** WindowedStream → DataStream AllWindowedStream → DataStream | Applies a general function to the window as a whole. Below is a function that manually sums the elements of a window.  **Note:** If you are using a windowAll transformation, you need to use an AllWindowFunction instead.  windowedStream**.**apply **(new** WindowFunction**<**Tuple2**<**String**,**Integer**>,** Integer**,** Tuple**,** Window**>()** **{**  **public** **void** **apply** **(**Tuple tuple**,**  Window window**,**  Iterable**<**Tuple2**<**String**,** Integer**>>** values**,**  Collector**<**Integer**>** out**)** **throws** Exception **{**  **int** sum **=** 0**;**  **for** **(**value t: values**)** **{**  sum **+=** t**.**f1**;**  **}**  out**.**collect **(new** **Integer(**sum**));**  **}**  **});**  *// applying an AllWindowFunction on non-keyed window stream*  allWindowedStream**.**apply **(new** AllWindowFunction**<**Tuple2**<**String**,**Integer**>,** Integer**,** Window**>()** **{**  **public** **void** **apply** **(**Window window**,**  Iterable**<**Tuple2**<**String**,** Integer**>>** values**,**  Collector**<**Integer**>** out**)** **throws** Exception **{**  **int** sum **=** 0**;**  **for** **(**value t: values**)** **{**  sum **+=** t**.**f1**;**  **}**  out**.**collect **(new** **Integer(**sum**));**  **}**  **});** |
| **Window Reduce** WindowedStream → DataStream | Applies a functional reduce function to the window and returns the reduced value.  windowedStream**.**reduce **(new** ReduceFunction**<**Tuple2**<**String**,**Integer**>()** **{**  **public** Tuple2**<**String**,** Integer**>** **reduce(**Tuple2**<**String**,** Integer**>** value1**,** Tuple2**<**String**,** Integer**>** value2**)** **throws** Exception **{**  **return** **new** Tuple2**<**String**,**Integer**>(**value1**.**f0**,** value1**.**f1 **+** value2**.**f1**);**  **}**  **};** |
| **Window Fold** WindowedStream → DataStream | Applies a functional fold function to the window and returns the folded value. The example function, when applied on the sequence (1,2,3,4,5), folds the sequence into the string "start-1-2-3-4-5":  windowedStream**.**fold**(**"start"**,** **new** FoldFunction**<**Integer**,** String**>()** **{**  **public** String **fold(**String current**,** Integer value**)** **{**  **return** current **+** "-" **+** value**;**  **}**  **};** |
| **Aggregations on windows** WindowedStream → DataStream | Aggregates the contents of a window. The difference between min and minBy is that min returns the minimun value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).  windowedStream**.**sum**(**0**);**  windowedStream**.**sum**(**"key"**);**  windowedStream**.**min**(**0**);**  windowedStream**.**min**(**"key"**);**  windowedStream**.**max**(**0**);**  windowedStream**.**max**(**"key"**);**  windowedStream**.**minBy**(**0**);**  windowedStream**.**minBy**(**"key"**);**  windowedStream**.**maxBy**(**0**);**  windowedStream**.**maxBy**(**"key"**);** |
| **Union** DataStream\* → DataStream | Union of two or more data streams creating a new stream containing all the elements from all the streams. Node: If you union a data stream with itself you will get each element twice in the resulting stream.  dataStream**.**union**(**otherStream1**,** otherStream2**,** **...);** |
| **Window Join** DataStream,DataStream → DataStream | Join two data streams on a given key and a common window.  dataStream**.**join**(**otherStream**)**  **.**where**(**0**).**equalTo**(**1**)**  **.**window**(**TumblingEventTimeWindows**.**of**(**Time**.**seconds**(**3**)))**  **.**apply **(new** **JoinFunction** **()** **{...});** |
| **Window CoGroup** DataStream,DataStream → DataStream | Cogroups two data streams on a given key and a common window.  dataStream**.**coGroup**(**otherStream**)**  **.**where**(**0**).**equalTo**(**1**)**  **.**window**(**TumblingEventTimeWindows**.**of**(**Time**.**seconds**(**3**)))**  **.**apply **(new** **CoGroupFunction** **()** **{...});** |
| **Connect** DataStream,DataStream → ConnectedStreams | "Connects" two data streams retaining their types. Connect allowing for shared state between the two streams.  DataStream**<**Integer**>** someStream **=** *//...*  DataStream**<**String**>** otherStream **=** *//...*  ConnectedStreams**<**Integer**,** String**>** connectedStreams **=** someStream**.**connect**(**otherStream**);** |
| **CoMap, CoFlatMap** ConnectedStreams → DataStream | Similar to map and flatMap on a connected data stream  connectedStreams**.**map**(new** CoMapFunction**<**Integer**,** String**,** Boolean**>()** **{**  @Override  **public** Boolean **map1(**Integer value**)** **{**  **return** **true;**  **}**  @Override  **public** Boolean **map2(**String value**)** **{**  **return** **false;**  **}**  **});**  connectedStreams**.**flatMap**(new** CoFlatMapFunction**<**Integer**,** String**,** String**>()** **{**  @Override  **public** **void** **flatMap1(**Integer value**,** Collector**<**String**>** out**)** **{**  out**.**collect**(**value**.**toString**());**  **}**  @Override  **public** **void** **flatMap2(**String value**,** Collector**<**String**>** out**)** **{**  **for** **(**String word: value**.**split**(**" "**))** **{**  out**.**collect**(**word**);**  **}**  **}**  **});** |
| **Split** DataStream → SplitStream | Split the stream into two or more streams according to some criterion.  SplitStream**<**Integer**>** split **=** someDataStream**.**split**(new** OutputSelector**<**Integer**>()** **{**  @Override  **public** Iterable**<**String**>** **select(**Integer value**)** **{**  List**<**String**>** output **=** **new** ArrayList**<**String**>();**  **if** **(**value **%** 2 **==** 0**)** **{**  output**.**add**(**"even"**);**  **}**  **else** **{**  output**.**add**(**"odd"**);**  **}**  **return** output**;**  **}**  **});** |
| **Select** SplitStream → DataStream | Select one or more streams from a split stream.  SplitStream**<**Integer**>** split**;**  DataStream**<**Integer**>** even **=** split**.**select**(**"even"**);**  DataStream**<**Integer**>** odd **=** split**.**select**(**"odd"**);**  DataStream**<**Integer**>** all **=** split**.**select**(**"even"**,**"odd"**);** |
| **Iterate** DataStream → IterativeStream → DataStream | Creates a "feedback" loop in the flow, by redirecting the output of one operator to some previous operator. This is especially useful for defining algorithms that continuously update a model. The following code starts with a stream and applies the iteration body continuously. Elements that are greater than 0 are sent back to the feedback channel, and the rest of the elements are forwarded downstream. See [iterations](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#iterations) for a complete description.  IterativeStream**<**Long**>** iteration **=** initialStream**.**iterate**();**  DataStream**<**Long**>** iterationBody **=** iteration**.**map **(***/\*do something\*/***);**  DataStream**<**Long**>** feedback **=** iterationBody**.**filter**(new** FilterFunction**<**Long**>(){**  @Override  **public** **boolean** **filter(**Integer value**)** **throws** Exception **{**  **return** value **>** 0**;**  **}**  **});**  iteration**.**closeWith**(**feedback**);**  DataStream**<**Long**>** output **=** iterationBody**.**filter**(new** FilterFunction**<**Long**>(){**  @Override  **public** **boolean** **filter(**Integer value**)** **throws** Exception **{**  **return** value **<=** 0**;**  **}**  **});** |
| **Extract Timestamps** DataStream → DataStream | Extracts timestamps from records in order to work with windows that use event time semantics. See [Event Time](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_time.html).  stream**.**assignTimestamps **(new** **TimeStampExtractor()** **{...});** |

|  |  |
| --- | --- |
| **Map** | Takes one element and produces one element.  data**.**map**(new** MapFunction**<**String**,** Integer**>()** **{**  **public** Integer **map(**String value**)** **{** **return** Integer**.**parseInt**(**value**);** **}**  **});** |
| **FlatMap** | Takes one element and produces zero, one, or more elements.  data**.**flatMap**(new** FlatMapFunction**<**String**,** String**>()** **{**  **public** **void** **flatMap(**String value**,** Collector**<**String**>** out**)** **{**  **for** **(**String s **:** value**.**split**(**" "**))** **{**  out**.**collect**(**s**);**  **}**  **}**  **});** |
| **MapPartition** | Transforms a parallel partition in a single function call. The function gets the partition as an Iterablestream and can produce an arbitrary number of result values. The number of elements in each partition depends on the degree-of-parallelism and previous operations.  data**.**mapPartition**(new** MapPartitionFunction**<**String**,** Long**>()** **{**  **public** **void** **mapPartition(**Iterable**<**String**>** values**,** Collector**<**Long**>** out**)** **{**  **long** c **=** 0**;**  **for** **(**String s **:** values**)** **{**  c**++;**  **}**  out**.**collect**(**c**);**  **}**  **});** |
| **Filter** | Evaluates a boolean function for each element and retains those for which the function returns true. **IMPORTANT:** The system assumes that the function does not modify the elements on which the predicate is applied. Violating this assumption can lead to incorrect results.  data**.**filter**(new** FilterFunction**<**Integer**>()** **{**  **public** **boolean** **filter(**Integer value**)** **{** **return** value **>** 1000**;** **}**  **});** |
| **Reduce** | Combines a group of elements into a single element by repeatedly combining two elements into one. Reduce may be applied on a full data set, or on a grouped data set.  data**.**reduce**(new** ReduceFunction**<**Integer**>** **{**  **public** Integer **reduce(**Integer a**,** Integer b**)** **{** **return** a **+** b**;** **}**  **});** |
| **ReduceGroup** | Combines a group of elements into one or more elements. ReduceGroup may be applied on a full data set, or on a grouped data set.  data**.**reduceGroup**(new** GroupReduceFunction**<**Integer**,** Integer**>** **{**  **public** **void** **reduce(**Iterable**<**Integer**>** values**,** Collector**<**Integer**>** out**)** **{**  **int** prefixSum **=** 0**;**  **for** **(**Integer i **:** values**)** **{**  prefixSum **+=** i**;**  out**.**collect**(**prefixSum**);**  **}**  **}**  **});**  If the reduce was applied to a grouped data set, you can specify the way that the runtime executes the combine phase of the reduce via supplying a CombineHint as a second parameter. The hash-based strategy should be faster in most cases, especially if the number of different keys is small compared to the number of input elements (eg. 1/10). |
| **Aggregate** | Aggregates a group of values into a single value. Aggregation functions can be thought of as built-in reduce functions. Aggregate may be applied on a full data set, or on a grouped data set.  Dataset**<**Tuple3**<**Integer**,** String**,** Double**>>** input **=** *// [...]*  DataSet**<**Tuple3**<**Integer**,** String**,** Double**>>** output **=** input**.**aggregate**(**SUM**,** 0**).**and**(**MIN**,** 2**);**  You can also use short-hand syntax for minimum, maximum, and sum aggregations.  Dataset**<**Tuple3**<**Integer**,** String**,** Double**>>** input **=** *// [...]*  DataSet**<**Tuple3**<**Integer**,** String**,** Double**>>** output **=** input**.**sum**(**0**).**andMin**(**2**);** |
| **Distinct** | Returns the distinct elements of a data set. It removes the duplicate entries from the input DataSet, with respect to all fields of the elements, or a subset of fields.  data**.**distinct**();** |
| **Join** | Joins two data sets by creating all pairs of elements that are equal on their keys. Optionally uses a JoinFunction to turn the pair of elements into a single element, or a FlatJoinFunction to turn the pair of elements into arbitrarily many (including none) elements. See the [keys section](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html#specifying-keys) to learn how to define join keys.  result **=** input1**.**join**(**input2**)**  **.**where**(**0**)** *// key of the first input (tuple field 0)*  **.**equalTo**(**1**);** *// key of the second input (tuple field 1)*  You can specify the way that the runtime executes the join via *Join Hints*. The hints describe whether the join happens through partitioning or broadcasting, and whether it uses a sort-based or a hash-based algorithm. Please refer to the [Transformations Guide](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/dataset_transformations.html#join-algorithm-hints) for a list of possible hints and an example. If no hint is specified, the system will try to make an estimate of the input sizes and pick the best strategy according to those estimates.  *// This executes a join by broadcasting the first data set*  *// using a hash table for the broadcasted data*  result **=** input1**.**join**(**input2**,** JoinHint**.**BROADCAST\_HASH\_FIRST**)**  **.**where**(**0**).**equalTo**(**1**);**  Note that the join transformation works only for equi-joins. Other join types need to be expressed using OuterJoin or CoGroup. |
| **OuterJoin** | Performs a left, right, or full outer join on two data sets. Outer joins are similar to regular (inner) joins and create all pairs of elements that are equal on their keys. In addition, records of the "outer" side (left, right, or both in case of full) are preserved if no matching key is found in the other side. Matching pairs of elements (or one element and a null value for the other input) are given to a JoinFunction to turn the pair of elements into a single element, or to a FlatJoinFunction to turn the pair of elements into arbitrarily many (including none) elements. See the [keys section](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html#specifying-keys) to learn how to define join keys.  input1**.**leftOuterJoin**(**input2**)** *// rightOuterJoin or fullOuterJoin for right or full outer joins*  **.**where**(**0**)** *// key of the first input (tuple field 0)*  **.**equalTo**(**1**)** *// key of the second input (tuple field 1)*  **.**with**(new** JoinFunction**<**String**,** String**,** String**>()** **{**  **public** String **join(**String v1**,** String v2**)** **{**  *// NOTE:*  *// - v2 might be null for leftOuterJoin*  *// - v1 might be null for rightOuterJoin*  *// - v1 OR v2 might be null for fullOuterJoin*  **}**  **});** |
| **CoGroup** | The two-dimensional variant of the reduce operation. Groups each input on one or more fields and then joins the groups. The transformation function is called per pair of groups. See the [keys section](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html#specifying-keys) to learn how to define coGroup keys.  data1**.**coGroup**(**data2**)**  **.**where**(**0**)**  **.**equalTo**(**1**)**  **.**with**(new** CoGroupFunction**<**String**,** String**,** String**>()** **{**  **public** **void** **coGroup(**Iterable**<**String**>** in1**,** Iterable**<**String**>** in2**,** Collector**<**String**>** out**)** **{**  out**.**collect**(...);**  **}**  **});** |
| **Cross** | Builds the Cartesian product (cross product) of two inputs, creating all pairs of elements. Optionally uses a CrossFunction to turn the pair of elements into a single element  DataSet**<**Integer**>** data1 **=** *// [...]*  DataSet**<**String**>** data2 **=** *// [...]*  DataSet**<**Tuple2**<**Integer**,** String**>>** result **=** data1**.**cross**(**data2**);**  Note: Cross is potentially a **very** compute-intensive operation which can challenge even large compute clusters! It is advised to hint the system with the DataSet sizes by using *crossWithTiny()* and*crossWithHuge()*. |
| **Union** | Produces the union of two data sets.  DataSet**<**String**>** data1 **=** *// [...]*  DataSet**<**String**>** data2 **=** *// [...]*  DataSet**<**String**>** result **=** data1**.**union**(**data2**);** |
| **Rebalance** | Evenly rebalances the parallel partitions of a data set to eliminate data skew. Only Map-like transformations may follow a rebalance transformation.  DataSet**<**String**>** in **=** *// [...]*  DataSet**<**String**>** result **=** in**.**rebalance**()**  **.**map**(new** **Mapper());** |
| **Hash-Partition** | Hash-partitions a data set on a given key. Keys can be specified as position keys, expression keys, and key selector functions.  DataSet**<**Tuple2**<**String**,**Integer**>>** in **=** *// [...]*  DataSet**<**Integer**>** result **=** in**.**partitionByHash**(**0**)**  **.**mapPartition**(new** **PartitionMapper());** |
| **Range-Partition** | Range-partitions a data set on a given key. Keys can be specified as position keys, expression keys, and key selector functions.  DataSet**<**Tuple2**<**String**,**Integer**>>** in **=** *// [...]*  DataSet**<**Integer**>** result **=** in**.**partitionByRange**(**0**)**  **.**mapPartition**(new** **PartitionMapper());** |
| **Custom Partitioning** | Manually specify a partitioning over the data.  *Note*: This method works only on single field keys.  DataSet**<**Tuple2**<**String**,**Integer**>>** in **=** *// [...]*  DataSet**<**Integer**>** result **=** in**.**partitionCustom**(**Partitioner**<**K**>** partitioner**,** key**)** |
| **Sort Partition** | Locally sorts all partitions of a data set on a specified field in a specified order. Fields can be specified as tuple positions or field expressions. Sorting on multiple fields is done by chaining sortPartition() calls.  DataSet**<**Tuple2**<**String**,**Integer**>>** in **=** *// [...]*  DataSet**<**Integer**>** result **=** in**.**sortPartition**(**1**,** Order**.**ASCENDING**)**  **.**mapPartition**(new** **PartitionMapper());** |
| **First-n** | Returns the first n (arbitrary) elements of a data set. First-n can be applied on a regular data set, a grouped data set, or a grouped-sorted data set. Grouping keys can be specified as key-selector functions or field position keys.  DataSet**<**Tuple2**<**String**,**Integer**>>** in **=** *// [...]*  *// regular data set*  DataSet**<**Tuple2**<**String**,**Integer**>>** result1 **=** in**.**first**(**3**);**  *// grouped data set*  DataSet**<**Tuple2**<**String**,**Integer**>>** result2 **=** in**.**groupBy**(**0**)**  **.**first**(**3**);**  *// grouped-sorted data set*  DataSet**<**Tuple2**<**String**,**Integer**>>** result3 **=** in**.**groupBy**(**0**)**  **.**sortGroup**(**1**,** Order**.**ASCENDING**)**  **.**first**(**3**);** |