Streams and Operations on Streams

NOTE: This document is a design draft and does not reflect the state of a stable release, but the work in progress on the current master, and future considerations.

Below is an overview of what types of streams are available in the system, and how they relate to each other.

DataStream

The DataStream is the core structure Flink's data stream API. It represents a parallel stream running in multiple stream partitions.

A DataStream is created from the StreamExecutionEnvironment via env.createStream(SourceFunction) (previously addSource(SourceFunction)).

Basic transformations on the data stream are record-at-a-time functions like map(), flatMap(), filter(). The data stream can be rebalanced (to mitigate skew) and broadcasted.

The records in a stream follow no particular order, but preserve the order as long as only record-at-a-time operations are applied.

Example:
DataStream Example

```scala
dataStream: DataStream[MyType] = env.createStream(new KafkaSource(...))
dataStream1: DataStream[(String, MyType)] = dataStream.flatMap(...)
dataStream2: DataStream[(String, MyType)] = dataStream2.rebalance()
dataStream3: DataStream[AnotherType] = dataStream3.map(...)
```

**DataStream Example**

```
dataStream: DataStream[MyType] = env.createStream(new KafkaSource(...))
dataStream1: DataStream[(String, MyType)] = dataStream.flatMap(...)
dataStream2: DataStream[(String, MyType)] = dataStream2.rebalance()
dataStream3: DataStream[AnotherType] = dataStream3.map(...)
```

---

**Stream Partition:** A stream partition is the stream of elements that originates at one parallel operator instance, and goes to one or more target operators. In the above example, a stream partition connects for example the first parallel instance of the source ($S_1$) and the first parallel instance of the `flatMap()` function ($m_1$). Another example of a stream partition is the stream from the first parallel instance of the `flatMap()` function ($m_1$) to all of the `map()` functions. In the latter case, the stream partition has multiple sub-partitions.

**Stream Sub-partition:** In cases where a stream partition has multiple receiving operator instances, it contains of multiple sub-partitions. In the above example, the stream between the `flatMap()` and `map()` function, due to the re-balancing. The data stream here has three partitions (originating at the parallel `flatMap()` instances), each of which has three sub-partitions (targeting the different `map()` instances).

**Note:** Those familiar with Apache Kafka, can think of a "stream" as a "Kafka topic" and of a "stream partition" as a "topic partition".

---

**KeyedDataStream**

A `KeyedDataStream` represents a data stream where elements are evaluated as "grouped" by a specified key. The `KeyedDataStream` serves two purposes:

- It is the first step in building a window stream, on top of which the grouped/windowed aggregation and reduce-style function can be applied.
- It allows to use the "by-key" state of functions. Here, every record has access to a state that is scoped by its key. Key-scoped state can be automatically redistributed and repartitioned.

A `KeyedDataStream` is typically created from a `DataStream` via calling `DataStream.keyBy()`. It is possible to directly create a `KeyedDataStream` from a source, if the source runs in parallel partitions (such as for example Apache Kafka) and the records are distributed over the partitions by key. The source distributes the elements across partitions, and the `KeySelector` defined what part of the element is the key (in the usual way: field name or selector function).

Any transformation function on a `KeyedDataStream` turns it into a `DataStream`, because the transformation may alter or remove the key, in the general case. This is a strong limitation, as all functions that could preserve the `KeyedDataStream` are record-at-a-time functions, which can be easily collapsed into one single `map()` or `flatMap()` function by the user.

---

**Example:**
KeyedDataStream Example

```scala
val stream: DataStream[MyType] = ...
val keyed: KeyedDataStream[MyType] = stream.keyBy("userId")
val countSoFar = keyed.map(new RichMapFunction[MyType, Int](){
  val state: OperatorState[Int] = _
  override def open(config: Configuration): Unit = {
    state = getRuntimeContext().getState("counts", 0, false)
  }
  override def map(in: MyType): Int = {
    val countSoFar = state.value + 1
    state.update(countSoFar)
    countSoFar
  }
})
```

WindowedDataStream

In a WindowedDataStream, records in the stream are organized into groups by the key, and per group, they are windowed together by the windowing. A WindowedDataStream is always created from a KeyedDataStream that assigns records to groups.

Example:

```scala
val stream: DataStream[MyType] = ...
val windowed: WindowedDataStream[MyType] = stream.keyBy("userId")
  .window(Count.of(4))
val result: DataStream[ResultType] = windowed.reduceGroup(myReducer)
```

Windows are maintained and evaluated for each group individually. For time, all groups will naturally have their windows evaluated at the same time. Evaluating windows locally per group makes the window coordination efficient, as no triggering events need to be coordinated across machines.

It is also important to notice that windows are in general not pre-materialized in a dedicated operator. They are built in the grouping operator as the groups are materialized.

Flink supports the following window types:

- Time - This may be operator time, ingress time, or event time (see Time and Order in Streams for details)
• Session - Session windows close after no record arrived for a group in a certain time. That time may be operator time, ingress time, or event time (see Time and Order in Streams for details)
• Count
• User-defined (via eviction / trigger)

For certain windows, the time of the elements may be important to assign semantics to the windows. For example, for a count window, it may make sense to define it takes windows of n records in the order of their timestamps. Unless such an order is requested, the elements are windows based on their arrival time (operator time). To respect record time, one would use a window as `DataStream.keyBy(field).window(Count.of(100), ElementOrder.ORDERED).reduce(myReducer)`.

**NonParallelWindowStream**

The NonParallelWindowStream is a means to run a windowed computation over all elements of the data stream, rather than over the elements of a group (by key). An example is the total number of records in the data stream so far, sliding by time. The NonParallelWindowedStream is non parallel, because in the general case, the window transformation or aggregate function needs all elements from the window. To avoid confusion and accidental bottlenecks, this stream is marked very explicitly as non-parallel.

The windowing/aggregating task needs only to maintain a single element buffer (or running aggregation element), because there is only one window and no groups.

**Pre-aggregation on time windows**

Windows on time can be efficiently pre-aggregated (if the aggregation/reduce function permits that), because time (as a global characteristic) can be evaluated at the pre-aggregators independently.

Example:

<table>
<thead>
<tr>
<th>Without pre-aggregation</th>
<th>With pre-aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>... maxBy() print()</td>
<td>... maxBy() maxBy() print()</td>
</tr>
<tr>
<td>Partition 1 o1</td>
<td>Partition 1 p1 o1 pa1</td>
</tr>
<tr>
<td>Partition 2 o2 a1 p2</td>
<td>Partition 2 p2 pa2 a1</td>
</tr>
<tr>
<td>Partition 3 o3</td>
<td>Partition 3 p3 pa3</td>
</tr>
</tbody>
</table>

**OrderedKeyedDataStream**

In the general case, the elements in a DataStream or a KeyedDataStream follow no specific order. In some cases, it is important that functions observe the records in a chronological order, with respect to the time that the records were produced at.

The OrderedKeyedDataStream can be used to give that guarantee. It buffers elements and brings them in order with respect to their timestamp. See Time and Order in Streams for details on this operation, including how timestamps of elements are defined, and how order is established.

Note that bringing a data stream in order usually means an increased latency, because functions can only be applied on the record once it is guaranteed that no earlier record may come any more. The amount of latency increase depends on how much the records are "out-of-order".
NOTE: Because stream partitions preserve their input order, operations directly after sources see the records in the order that the sources emit them. For some sources (such as Apache Kafka), this may be well defined.

Joins

Joins are initially supported only on equi-sized tumbling time windows. Since time is a global characteristic, it has an equal meaning for both inputs and it is guaranteed to equally and continuously progress in both inputs.

The fact that time always makes progress is important: A window definition where one input's window would not make progress for a while would factually stop the join and stop the stream on the other input as well.

The window time may be defined as event time (see Time and Order in Streams). These joins are well defined, parallelizable, and can compute windows and join data structures together (as in the case of grouped windows).

Later additions may add sliding time windows.

![Diagram of join implementation]

**DataStream Example**

```scala
val firstInput: DataStream[MyType] = ...
val secondInput: DataStream[AnotherType] = ...
val firstKeyed = firstInput.keyBy("userId")
val secondKeyed = secondInput.keyBy("id")
val result: DataStream[(MyType, AnotherType)] =
  firstKeyed.join(secondKeyed)
onWindow(Time.of(5, SECONDS))
```

Join implementations:

- Simple hash join that builds a hash table on one side (until the window is full) and buffers data on the other input. Once the window is complete, the buffered data is joined against the window.
  - Elements come in order in cases where multiple windows are built.
  - Latency after window is triggered

- Symmetric hash join implementation
  - Low latency after window is triggered
  - May be more compute intensive to maintain two hash tables
  - Harder to add out-of-core functionality
  - When elements come out-of-order, result is out-of-order.

**ConnectedDataStream**
The ConnectedDataStream is a way to share state between two tuple-at-a-time operations. It can be through as executing two MapFunctions (or FlatMapFunctions) in the same object. For an initial consistent version, this operation should be restricted to "record-at-a-time" operations only (effectively map() and flatMap()). Since any form of grouping operation requires a window, doing this operation on groups would lead to very high complexity.

![Diagram of ConnectedDataStream]

**DataStream Example**

```scala
val input: DataStream[MyType] = ...
val other: DataStream[AnotherType] = ...
val connected: ConnectedDataStream[MyType, AnotherType] = input.connect(other)
val result: DataStream[ResultType] = connected.coMap(new CoMapFunction[MyType, AnotherType, ResultType]() {
  override def map1(value: MyType): ResultType = { ... }
  override def map2(value: AnotherType): ResultType = { ... }
})
```

**Discretization**

A DataStream can be discretized by time, in which case it becomes a DiscretizedStream which is a sequence of DataSets. DiscretizedStreams support the usual data set style operations by transforming each DataSet in the sequence, which always result again in a DiscretizedStream. This concept is very similar to Spark RDDs. The DiscretizedStream can be converted back to a DataStream by calling DiscretizedStream.toStream().

The discretization can happen by all forms of time described under Time and Order in Streams. Discretization cannot happen by any other characteristic than time, because time is the only global characteristic that can be evaluated independently in parallel.

**Discretization Example**

```scala
val stream: DataStream[MyType] = ...;
val sets: DiscretizedStream[MyType] = stream.discretize(EventTime.of(1, MINUTE))

//Define transformations on the individual DataSets
val someSet: DiscretizedStream[MyType] = sets.filter(...)
val anotherSet: DiscretizedStream[MyType] = sets.foreach {
  set => set.map(...).groupBy(...).avg(...)
}

//Pairwise (zip) join the data sets in the DiscretizedStream
val result: DiscretizedStream[(MyType, MyType)] = someSet.join(anotherSet).where(x).equalTo(y)
val resultStream: DataStream[(MyType, MyType)] = result.toStream()
resultStream.keyBy(...).window(Count.of(10)).reduce(...)
```

**NOTE:** For later, we may think about adding the possibility to transform non-parallel windows into a data set, which would allow other forms of discretization, that result in a data set with parallelism 1 (from which later operations could be parallel again).

**Interactions between DiscretizedStreams**
It is necessary for the majority of applications to allow interactions between multiple DiscretizedStream. This interactions range from simple union operators to more complex operations like joins.

Interactions should only be possible between DiscretizedStreams that have the same discretization interval (same time scale and length) to make the behaviour intuitive. In this operators are applied on the time-aligned DataSets from each DiscretizedStream, practically zipping the DataSets and then applying the operations.

Applying operations on DiscretizedStreams with non-matching discretization should throw an exception before submitting the job to the cluster.

DiscretizedStream join example:

Interactions between DiscretizedStreams and DataSets
tbd

Window Pre-Aggregation

Panes for Time Windows

Invertible aggregations

Open Issues

Record locating themselves in windows

We currently have no design that allows records to find out in which window they are. This most likely makes sense only for time windows. Cloud Dataflow supports that feature. Is that desirable to add this to Flink in the long run?