IoT Big Data Processing

Apache Flink

Albert Bifet(@abifet)

Apache Flink Motivation

Apache Flink Motivation

1. Real time computation: streaming computation
2. Fast, as there is not need to write to disk
3. Easy to write code

Real time computation: streaming computation

MapReduce Limitations

Example

How compute in real time (latency less than 1 second):

1. frequent items as Twitter hashtags
2. predictions
3. sentiment analysis

Easy to Write Code

```
case class Word (word: String, frequency: Int)
```

DataSet API (batch):

```
val lines: DataSet[String] = env.readTextFile(...)

lines.flatMap {line => line.split(" ")
                            .map(word => Word(word,1))}
     .groupBy("word").sum("frequency")
     .print()
```

Easy to Write Code

```
case class Word (word: String, frequency: Int)
```

DataSet API (batch):

```
val lines: DataSet[String] = env.readTextFile(...)

lines.flatMap {line => line.split(" ")
                       .map(word => Word(word,1))}
    .groupBy("word").sum("frequency")
    .print()
```

DataStream API (streaming):

```
val lines: DataStream[String] = env.fromSocketStream(...)

lines.flatMap {line => line.split(" ")
                       .map(word => Word(word,1))}
    .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))
    .groupBy("word").sum("frequency")
    .print()
```

What is Apache Flink?

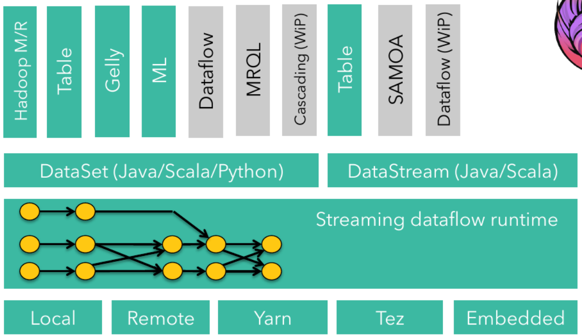


Figure: Apache Flink Overview

Batch and Streaming Engines

Batch only

Streaming only

Hybrid

Figure: Batch, streaming and hybrid data processing engines.

Batch Comparison

API	low-level	high-level	high-level
Data Transfer	batch	batch	pipelined & batch
Memory Management	disk-based	JVM-managed	Active managed
Iterations	file system cached	in-memory cached	streamed
Fault tolerance	task level	task level	job level
Good at	massive scale out	data exploration	heavy backend & iterative jobs
Libraries	many external	built-in & external	evolving built-in & external

Figure: Comparison between Hadoop, Spark And Flink.

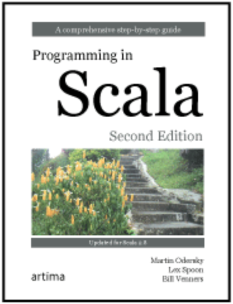
Streaming Comparison

Streaming	“true”	mini batches	“true”
API	low-level	high-level	high-level
Fault tolerance	tuple-level ACKs	RDD-based (lineage)	coarse checkpointing
State	not built-in	external	internal
Exactly once	at least once	exactly once	exactly once
Windowing	not built-in	restricted	flexible
Latency	low	medium	low
Throughput	medium	high	high

Figure: Comparison between Storm, Spark And Flink.

Scala Language

- What is Scala?
  - object oriented
  - functional
- How is Scala?
  - Scala is compatible
  - Scala is concise
  - Scala is high-level
  - Scala is statically typed



Short Course on Scala

- Easy to use: includes an interpreter
- Variables:
  - val: immutable (preferable)
  - var: mutable
- Scala treats everything as objects with methods
- Scala has first-class functions
- Functions:

```
def max(x: Int, y: Int): Int = {
    if (x > y) x
        else y
    }

def max2(x: Int, y: Int) = if (x > y) x else y
```

Short Course on Scala

- Functional:

```scala
args.foreach((arg: String) => println(arg))
args.foreach(arg => println(arg))
args.foreach(println)
```

- Imperative:

```scala
for (arg <- args) println(arg)
```

- Scala achieves a conceptual simplicity by treating everything, from arrays to expressions, as objects with methods.

```scala
(1).+(2)

greetStrings(0) = "Hello"
greetStrings.update(0, "Hello")
val numNames2 = Array.apply("zero", "one", "two")
```

Short Course on Scala

- **Array**: mutable sequence of objects that share the same type
- **List**: immutable sequence of objects that share the same type
- **Tuple**: immutable sequence of objects that does not share the same type

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

Short Course on Scala

- Sets and maps

```scala
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))

import scala.collection.mutable.Map
val treasureMap = Map[Int, String]()
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
println(treasureMap(2))
```

Short Course on Scala

- Functional style
    - Does not contain any var

```
def printArgs(args: Array[String]): Unit = {
    var i = 0
    while (i < args.length) {
        println(args(i))
        i += 1
    }
}

def printArgs(args: Array[String]): Unit = {
    for (arg <- args)
        println(arg)
}

def printArgs(args: Array[String]): Unit = {
    args.foreach(println)
}

def formatArgs(args: Array[String]) = args.mkString("\n")
println(formatArgs(args))
```

Short Course on Scala

- Prefer vals, immutable objects, and methods without side effects.
- Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.
- In a Scala program, a semicolon at the end of a statement is usually optional.
- A singleton object definition looks like a class definition, except instead of the keyword class you use the keyword object .
- Scala provides a trait, scala.Application:

```
object FallWinterSpringSummer extends Application {
  for (season <- List("fall", "winter", "spring"))
      println(season +": "+ calculate(season))
}
```

Short Course on Scala

- Scala has first-class functions: you can write down functions as unnamed literals and then pass them around as values.

```
(x: Int) => x + 1
```

- Short forms of function literals

```
someNumbers.filter((x: Int) => x > 0)
someNumbers.filter((x) => x > 0)
someNumbers.filter(x => x > 0)
someNumbers.filter(_ > 0)

someNumbers.foreach(x => println(x))
someNumbers.foreach(println _)
```

Short Course on Scala

- Zipping lists: zip and unzip
  - The zip operation takes two lists and forms a list of pairs:
  - A useful special case is to zip a list with its index. This is done most efficiently with the zipWithIndex method,
- Mapping over lists: map , flatMap and foreach
- Filtering lists: filter , partition , find , takeWhile , dropWhile , and span
- Folding lists: /: and : or foldLeft and foldRight.

  ```
  (z /: List(a, b, c)) (op)
  ```

  equals

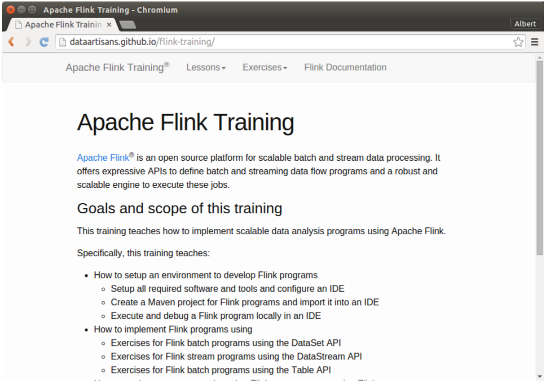
  ```
  op(op(op(z, a), b), c)
  ```

- Sorting lists: sortWith

Apache Flink Architecture

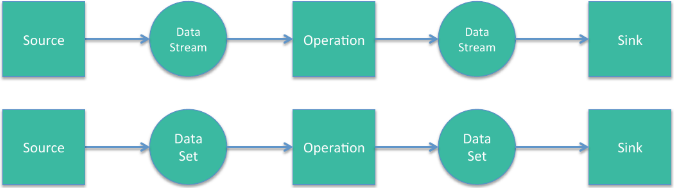
References

Apache Flink Documentation



`http://dataartisans.github.io/flink-training/`

API Introduction



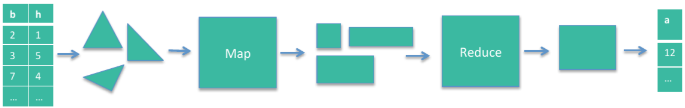
Flink programs

1. Input from source
2. Apply operations
3. Output to source

Batch and Streaming APIs

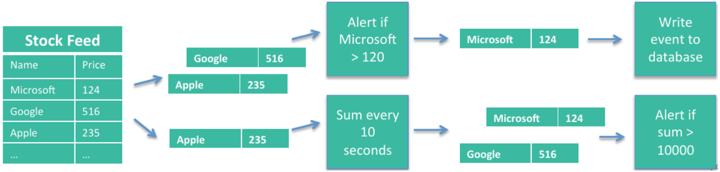
❶ DataSet API

- Example: Map/Reduce paradigm



❷ DataStream API

- Example: Live Stock Feed



Streaming and Batch Comparison

	Streaming	Batch
Input	infinite	finite
Data transfer	pipelined	blocking or pipelined
Latency	low	high

Architecture Overview

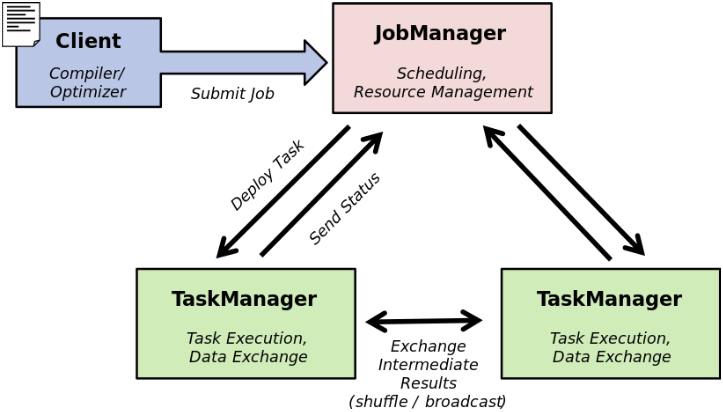
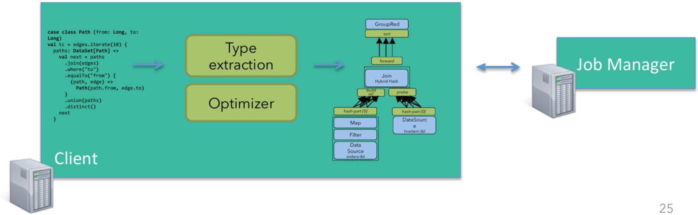


Figure: The JobManager is the coordinator of the Flink system
TaskManagers are the workers that execute parts of the parallel programs.

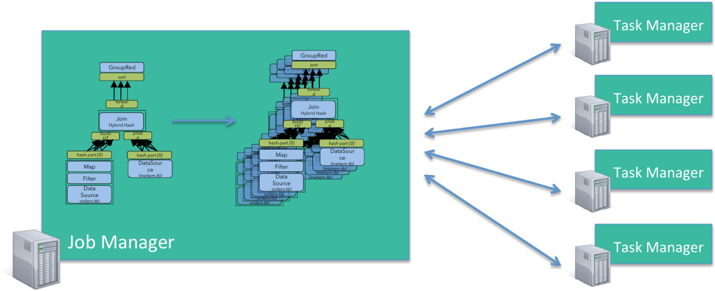
Client

1. Optimize
2. Construct job graph
3. Pass job graph to job manager
4. Retrieve job results



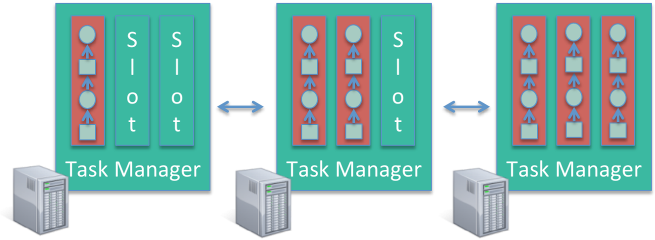
Job Manager

1. **Parallelization**: Create Execution Graph
2. **Scheduling**: Assign tasks to task managers
3. **State**: Supervise the execution

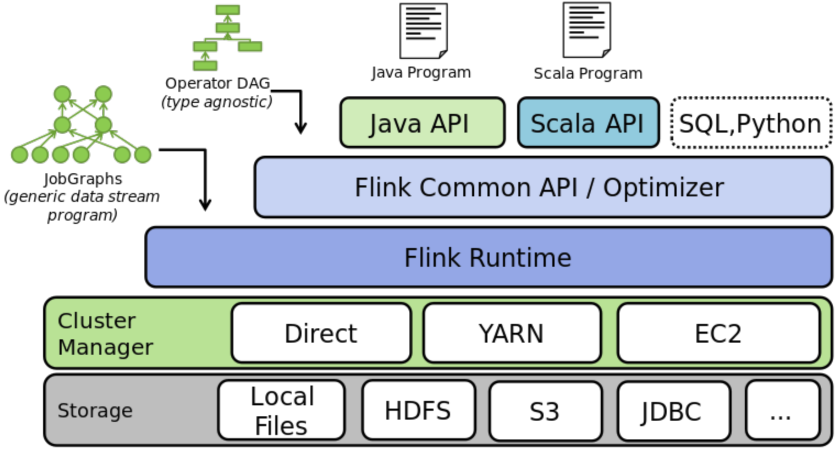


Task Manager

1. Operations are split up into **tasks** depending on the specified parallelism
2. Each parallel instance of an operation runs in a separate **task slot**
3. The scheduler may run several tasks from different operators in one task slot



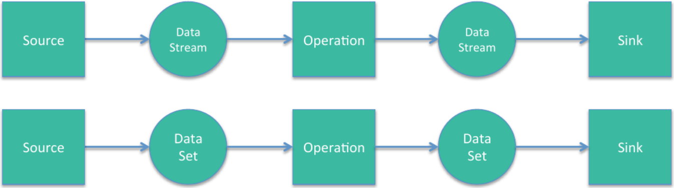
Component Stack



Component Stack

1. **API layer:** implements multiple APIs that create operator DAGs for their programs. Each API needs to provide utilities (serializers, comparators) that describe the interaction between its data types and the runtime.
2. **Optimizer and common api layer:** takes programs in the form of operator DAGs. The operators are specific (e.g., Map, Join, Filter, Reduce, . . .), but are data type agnostic.
3. **Runtime layer:** receives a program in the form of a JobGraph. A JobGraph is a generic parallel data flow with arbitrary tasks that consume and produce data streams.

Flink Topologies



Flink programs

1. Input from source
2. Apply operations
3. Output to source

Sources (selection)

- Collection-based
  - fromCollection
  - fromElements
- File-based
  - TextInputFormat
  - CsvInputFormat
- Other
  - SocketInputFormat
  - KafkaInputFormat
  - Databases

Sinks (selection)

- File-based
  - TextOutputFormat
  - CsvOutputFormat
  - PrintOutput
- Others
  - SocketOutputFormat
  - KafkaOutputFormat
  - Databases

Apache Flink Algorithns

Flink Skeleton Program

1. Obtain an `ExecutionEnvironment`,
2. Load/create the initial data,
3. Specify transformations on this data,
4. Specify where to put the results of your computations,
5. Trigger the program execution

Java WordCount Example

```
public class WordCountExample {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env =
            ExecutionEnvironment.getExecutionEnvironment();

        DataSet<String> text = env.fromElements(
            "Who's there?",
            "I think I hear them. Stand, ho! Who's there?");

        DataSet<Tuple2<String, Integer>> wordCounts = text
            .flatMap(new LineSplitter())
            .groupBy(0)
            .sum(1);

        wordCounts.print();
    }

    ....
}
```

Java WordCount Example

```
public class WordCountExample {
    public static void main(String[] args) throws Exception {
      ....
    }

    public static class LineSplitter implements
       FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String line, Collector<Tuple2<String,
                                             Integer>> out) {
            for (String word : line.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }
}
```

Scala WordCount Example

```scala
import org.apache.flink.api.scala._

object WordCount {
  def main(args: Array[String]) {

    val env = ExecutionEnvironment.getExecutionEnvironment
    val text = env.fromElements(
      "Who's there?",
      "I think I hear them. Stand, ho! Who's there?")

    val counts = text.flatMap
        { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
      .map { (_, 1) }
      .groupBy(0)
      .sum(1)

    counts.print()
  }
}
```

Java 8 WordCount Example

```
public class WordCountExample {
  public static void main(String[] args) throws Exception {
    final ExecutionEnvironment env =
          ExecutionEnvironment.getExecutionEnvironment();

    DataSet<String> text = env.fromElements(
        "Who's there?"
        "I think I hear them. Stand, ho! Who's there?");

    text.map(line -> line.split(" "))
    .flatMap((String[] wordArray,
              Collector<Tuple2<String, Integer>> out)
        -> Arrays.stream(wordArray)
                .forEach(t -> out.collect(new Tuple2<>(t, 1)))
        )
    .groupBy(0)
    .sum(1)
    .print();
  }
}
```

1) Obtain an `ExecutionEnvironment`

The `ExecutionEnvironment` is the basis for all Flink programs.

```
getExecutionEnvironment()

createLocalEnvironment()
createLocalEnvironment(int parallelism)
createLocalEnvironment(Configuration customConfiguration)

createRemoteEnvironment(String host, int port, String... jarFiles)
createRemoteEnvironment(String host, int port, int parallelism,
                        String... jarFiles)
```

2) Data sources: load/create the initial data

- File-based:
  - readTextFile(path) / TextInputFormat
  - readTextFileWithValue(path) / TextValueInputFormat
  - readCsvFile(path) / CsvInputFormat
  - readFileOfPrimitives(path, Class) / PrimitiveInputFormat
- Collection-based:
  - fromCollection(Collection)
  - fromCollection(Iterator, Class)
  - fromElements(T ...)
  - fromParallelCollection(SplittableIterator, Class)
  - generateSequence(from, to)
- Generic:
  - readFile(inputFormat, path) / FileInputFormat
  - createInput(inputFormat) / InputFormat

```
DataSet<String> text = env.readTextFile("file:///path/to/file");
```

3) Specify transformations on this data

Map

Takes one element and produces one element.

```
data.map { x => x.toInt }
```

3) Specify transformations on this data

FlatMap

Takes one element and produces zero, one, or more elements.

```
data.flatMap { str => str.split(" ") }
```

3) Specify transformations on this data

MapPartition

Transforms a parallel partition in a single function call. The function get the partition as an ‘Iterator‘ 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 { in => in map { (_, 1) } }
```

3) Specify transformations on this data

Filter

Evaluates a boolean function for each element and retains those for which the function returns true.

```
data.filter { _ > 1000 }
```

3) Specify transformations on this data

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 { _ + _ }
```

3) Specify transformations on this data

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 { elements => elements.sum }
```

3) Specify transformations on this data

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.

```
val input: DataSet[(Int, String, Double)] = // [...]
val output: DataSet[(Int, String, Double)] =
           input.aggregate(SUM, 0).aggregate(MIN, 2);
```

You can also use short-hand syntax for minimum, maximum, and sum aggregations.

```
val input: DataSet[(Int, String, Double)] = // [...]
val output: DataSet[(Int, String, Double)] = input.sum(0).min(2)
```

3) Specify transformations on this data

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.

```
data1.coGroup(data2).where(0).equalTo(1)
```

3) Specify transformations on this data

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

```
val data1: DataSet[Int] = // [...]
val data2: DataSet[String] = // [...]
val result: DataSet[(Int, String)] = data1.cross(data2)
```

3) Specify transformations on this data

Union

Produces the union of two data sets.

```
data.union(data2)
```

3) Specify transformations on this data

Rebalance

Evenly rebalances the parallel partitions of a data set to eliminate data skew. Only Map-like transformations may follow a rebalance transformation.

```
val data1: DataSet[Int] = // [...]
val result: DataSet[(Int, String)] = data1.rebalance().map(...)
```

3) Specify transformations on this data

Hash-Partition

Hash-partitions a data set on a given key. Keys can be specified as key-selector functions, tuple positions or case class fields.

```
val in: DataSet[(Int, String)] = // [...]
val result = in.partitionByHash(0).mapPartition { ... }
```

3) Specify transformations on this data

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.

```
val in: DataSet[(Int, String)] = // [...]
val result = in.sortPartition(1, Order.ASCENDING).mapPartition { ... }
```

3) Specify transformations on this data

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, tuple positions or case class fields.

```
val in: DataSet[(Int, String)] = // [...]
// regular data set
val result1 = in.first(3)
// grouped data set
val result2 = in.groupBy(0).first(3)
// grouped-sorted data set
val result3 = in.groupBy(0).sortGroup(1, Order.ASCENDING).first(3)
```

4) Data sinks: Specify where to put the results of your computations

- `writeAsText() / TextOuputFormat`
- `writeAsFormattedText() / TextOutputFormat`
- `writeAsCsv(...)  / CsvOutputFormat`
- `print() / printToErr() / print(String msg) / printToErr(String msg)`
- `write() / FileOutputFormat`
- `output()/ OutputFormat`

```
writeAsText(String path)
writeAsCsv(String path)
write(FileOutputFormat<T> outputFormat, String filePath)

print()
printOnTaskManager()

collect()
```

5) Trigger the program execution

- print() and collect() do not return‘ the result, but it can be accessed from the getLastJobExecutionResult() method
- execute() method returns the JobExecutionResult, including execution times and accumulator results.

Data Streams Algorithns

Flink Skeleton Program

1. Obtain an `StreamExecutionEnvironment`,
2. Load/create the initial data,
3. Specify transformations on this data,
4. Specify where to put the results of your computations,
5. Trigger the program execution

Java WordCount Example

```java
public class StreamingWordCount {

    public static void main(String[] args) {

        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Tuple2<String, Integer>> dataStream = env
                .socketTextStream("localhost", 9999)
                .flatMap(new Splitter())
                .groupBy(0)
                .sum(1);

        dataStream.print();

        env.execute("Socket Stream WordCount");
    }

    ....
}
```

Java WordCount Example

```
public class StreamingWordCount {
    public static void main(String[] args) throws Exception {
      ....
    }

   public static class Splitter implements
     FlatMapFunction<String, Tuple2<String, Integer>> {
       @Override
       public void flatMap(String sentence,
            Collector<Tuple2<String, Integer>> out) throws Exception
         for (String word: sentence.split(" ")) {
            out.collect(new Tuple2<String, Integer>(word, 1));
         }
       }
    }
}
```

Scala WordCount Example

```
object WordCount {
  def main(args: Array[String]) {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream("localhost", 9999)

    val counts = text.flatMap { _.toLowerCase.split("\\W+")
                                filter { _.nonEmpty } }
      .map { (_, 1) }
      .groupBy(0)
      .sum(1)

    counts.print

    env.execute("Scala Socket Stream WordCount")
  }
}
```

Obtain an `StreamExecutionEnvironment`

The `StreamExecutionEnvironment` is the basis for all Flink programs.

```
StreamExecutionEnvironment.getExecutionEnvironment
StreamExecutionEnvironment.createLocalEnvironment(parallelism)
StreamExecutionEnvironment.createRemoteEnvironment(host: String,
        port: String, parallelism: Int, jarFiles: String*)

env.socketTextStream(host, port)
env.fromElements(elements...)
env.addSource(sourceFunction)
```

Specify transformations on this data

- Map
- FlatMap
- Filter
- Reduce
- Fold
- Union

Window Operators

The user has different ways of using the result of a window operation:

- `windowedDataStream.flatten()` - streams the results element wise and returns a DataStream<T> where T is the type of the underlying windowed stream
- `windowedDataStream.getDiscretizedStream()` - returns a DataStream<StreamWindow<T>> for applying some advanced logic on the stream windows itself.
- Calling any window transformation further transforms the windows, while preserving the windowing logic

```
dataStream.window(Time.of(5, TimeUnit.SECONDS))
          .every(Time.of(1, TimeUnit.SECONDS));

dataStream.window(Count.of(100))
          .every(Time.of(1, TimeUnit.MINUTES));
```

Gelly: Flink Graph API

- Gelly is a Java Graph API for Flink.
- In Gelly, graphs can be transformed and modified using high-level functions similar to the ones provided by the batch processing API.
- In Gelly, a Graph is represented by a DataSet of vertices and a DataSet of edges.
- The Graph nodes are represented by the Vertex type. A Vertex is defined by a unique ID and a value.

```
// create a new vertex with a Long ID and a String value
Vertex<Long, String> v = new Vertex<Long, String>(1L, "foo");

// create a new vertex with a Long ID and no value
Vertex<Long, NullValue> v =
    new Vertex<Long, NullValue>(1L, NullValue.getInstance());
```

Gelly: Flink Graph API

- The graph edges are represented by the Edge type.
- An Edge is defined by a source ID (the ID of the source Vertex), a target ID (the ID of the target Vertex) and an optional value.
- The source and target IDs should be of the same type as the Vertex IDs. Edges with no value have a NullValue value type.

```
Edge<Long, Double> e = new Edge<Long, Double>(1L, 2L, 0.5);

// reverse the source and target of this edge
Edge<Long, Double> reversed = e.reverse();

Double weight = e.getValue(); // weight = 0.5
```

Table API - Relational Queries

- Flink provides an API that allows specifying operations using SQL-like expressions.
- Instead of manipulating DataSet or DataStream you work with Table on which relational operations can be performed.

```
import org.apache.flink.api.scala._
import org.apache.flink.api.scala.table._

case class WC(word: String, count: Int)
val input = env.fromElements(WC("hello", 1),
                  WC("hello", 1), WC("ciao", 1))
val expr = input.toTable
val result = expr.groupBy('word)
                 .select('word, 'count.sum as 'count).toDataSet[WC]
```