EPL646 – Advanced Topics in Databases
Apache Flink

http://www.cs.ucy.ac.cy/~dzeina/courses/epl646/labs/lab.html

University of Cyprus

DMSL @UCY

Introduction to Apache Flink

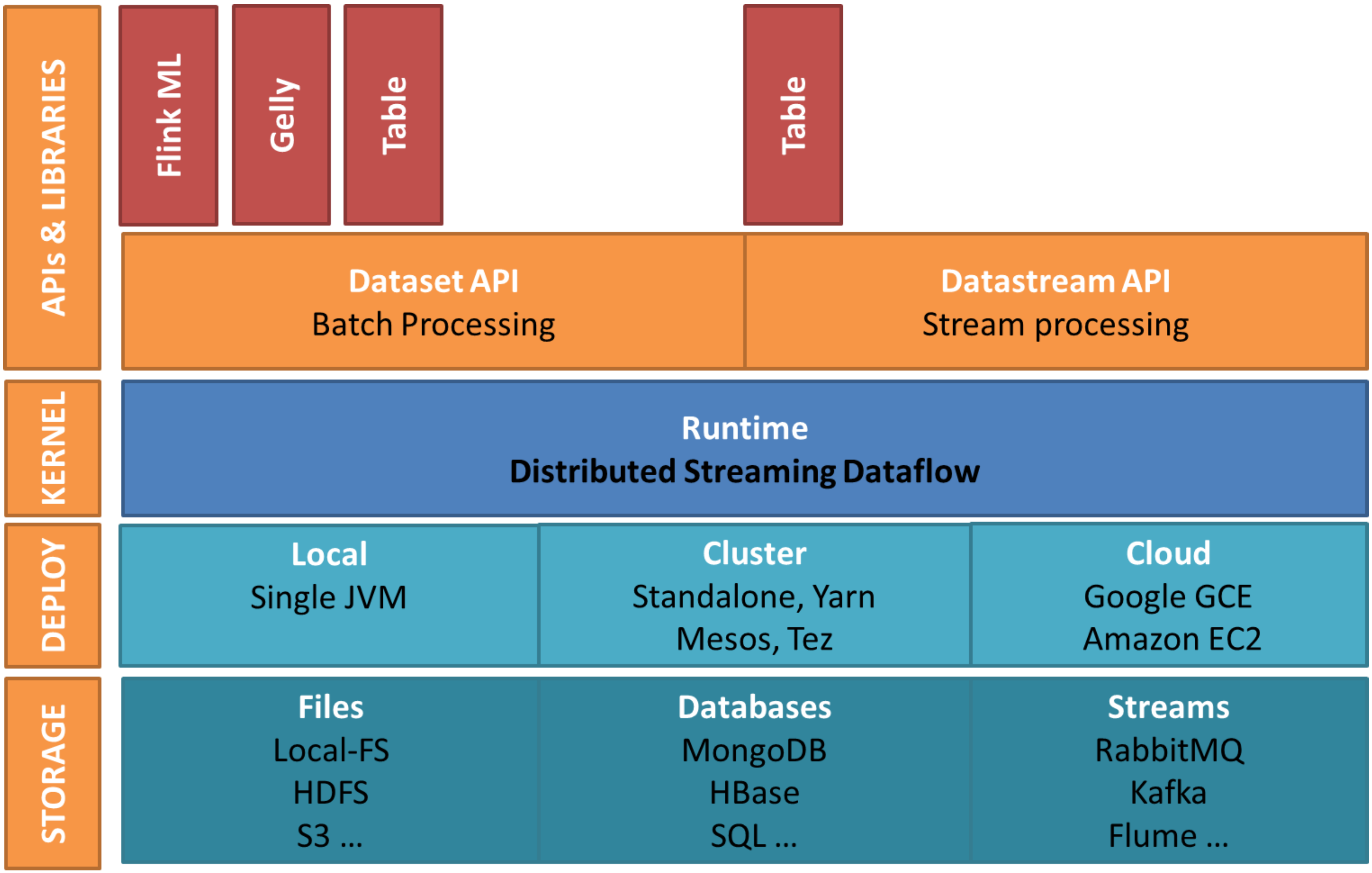
Apache Flink is the next generation Big Data tool also known as 4G of Big Data.

- It is the true stream processing framework
  - doesn't cut stream into micro-batches
- Flink's kernel (core) is a streaming runtime which also provides distributed processing, fault tolerance, etc.
- Flink processes events at a consistently high speed with low latency
- It processes the data at lightning fast speed
- It is the large-scale data processing framework which can process data generated at very high velocity

Introduction to Apache Flink

- Flink is an alternative to MapReduce, it processes data more than 100 times faster than MapReduce
- It is independent of Hadoop but it can use HDFS to read, write, store, process the data
- Flink does not provide its own data storage system. It takes data from distributed storage

Flink's Ecosystem



DataSet API

- Handles the data at the rest
- Allows the user to implement operations like map, filter, join, group, etc. on the dataset
- Mainly used for distributed processing
- It is a special case of Stream processing where we have a finite data source
  - The batch application is also executed on the streaming runtime

DataStream API

- Handles a continuous stream of the data
  - To process live data stream it provides various operations like map, filter, update states, window, aggregate, etc.

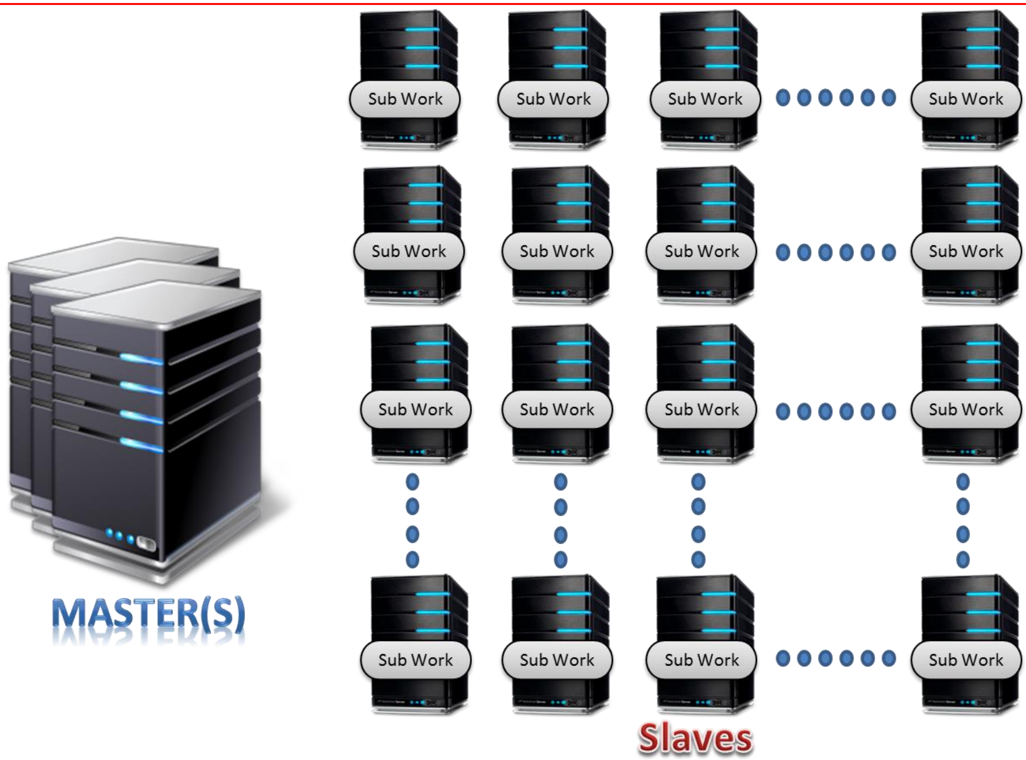
- Can consume the data from the various streaming source and can write the data to different sinks

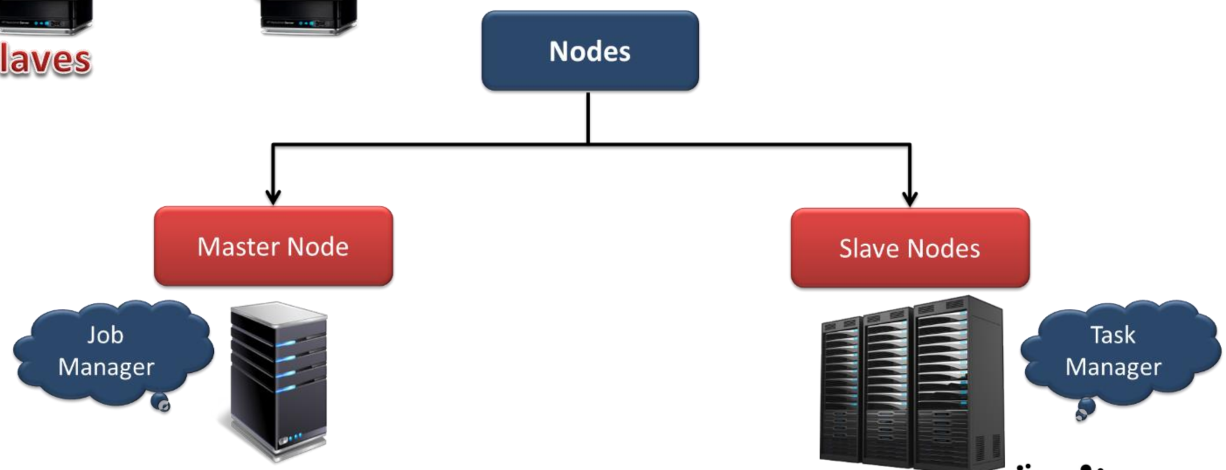
- Supports both **Java** and **Scala**.

Domain Specific Library Tool's

- Table
  - Enables users to perform ad-hoc analysis using SQL like expression language for relational stream and batch processing
  - Can be embedded in *DataSet* and *DataStream* APIs
  - Saves users from writing complex code to process the data instead allows them to run SQL queries on the top of Flink
- Gelly
  - Graph processing engine which allows users to run set of operations to create, transform and process the graph
  - Provides a library to simplify the development of graph applications
  - Available in **Java** and **Scala**.
- FlinkML
  - A machine learning library which provides intuitive APIs and an efficient algorithm to handle machine learning applications
  - Available in **Scala**.

Flink Architecture





Flink Features

- Streaming & Stream processing
- High performance
- Low latency
- Event Time and Out-of-Order Events
- Lightning fast speed
- Fault Tolerance
- Memory management
- Broad integration
- Program optimizer
- Scalable
- Rich set of operators
- Exactly-once Semantics
- Highly flexible Streaming Windows
- Continuous streaming model with backpressure
- One Runtime for Streaming and Batch Processing
- Easy and understandable Programmable APIs
- Little tuning required

Core API Concepts

- Every Flink program performs transformations on distributed collections of data
  - A variety of functions for transforming data are provided, including filtering, mapping, joining, grouping, and aggregating
- A sink operation in Flink triggers the execution of a stream to produce the desired result of the program
  - such as saving the result to the file system or printing it to the standard output
- Flink transformations are lazy
  - they are not executed until a sink operation is invoked
- The Apache Flink API supports two modes of operations: batch and real-time.
  - If you are dealing with a limited data source that can be processed in batch mode, you will use the *DataSet* API
  - Should you want to process unbounded streams of data in real time, you would need to use the *DataStream* API

DataSet API Transformations

- The entry point to the Flink program is an instance of the *ExecutionEnvironment* class
  - defines the context in which a program is executed

```
ExecutionEnvironment env =
    ExecutionEnvironment.getExecutionEnvironment();
```

Creating a DataSet

- To start performing data transformations, we need to supply our program with the data

```
DataSet<Integer> amounts =
        env.fromElements(1, 29, 40, 50);
```

- You can create a *DataSet* from multiple sources, such as Apache Kafka, a CSV, a file or virtually any other data source

Filter and Reduce

- Once you create an instance of the *DataSet* class, you can apply transformations to it
- Let's say that you want to filter numbers that are above a certain threshold and next sum them all. You can use the filter() and reduce() transformations to achieve this:

```
int threshold = 30;
List<Integer> collect = amounts
    .filter(a -> a > threshold)
    .reduce((integer, t1) -> integer + t1)
    .collect();
```

- Note that the collect() method is a sink operation that triggers the actual data transformations

Map

- Let’s say that you have a *DataSet* of Person objects:

```
private static class Person {
     private int age;
     private String name;
     // standard constructors/getters/setters
}
```

- Next, let’s create a *DataSet* of these objects:

```
DataSet<Person> personDataSource = env.fromCollection(
      Arrays.asList(
          new Person(23, "Tom"),
          new Person(75, "Michael")));
```

- Suppose that you want to extract only the age field from every object of the collection. You can use the map() transformation to get only a specific field of the Person class:

```
List<Integer> ages = personDataSource
     .map(p -> p.age)
     .collect();
```

Join

- When you have two datasets, you may want to join them on some id field
  - use the join() transformation
- Let's create collections of transactions and addresses of a user:

```
Tuple3<Integer, String, String> address =
  new Tuple3<>(1, "5th Avenue", "London");
DataSet<Tuple3<Integer, String, String>> addresses =
  env.fromElements(address);
Tuple2<Integer, String> firstTransaction =
  new Tuple2<>(1, "Transaction_1");
DataSet<Tuple2<Integer, String>> transactions =
  env.fromElements(firstTransaction,
                   new Tuple2<>(12, "Transaction_2"));
```

- The first field in both tuples is of an Integer type, and this is an id field on which we want to join both data sets.

Join

- To perform the actual joining logic, we need to implement a KeySelector interface for address and transaction:

```
private static class IdKeySelectorTransaction
    implements KeySelector<Tuple2<Integer, String>, Integer> {
    @Override
    public Integer getKey(Tuple2<Integer, String> value) {
      return value.f0;
    }
}
private static class IdKeySelectorAddress
    implements KeySelector<Tuple3<Integer, String, String>, Integer> {
    @Override
    public Integer getKey(Tuple3<Integer, String, String> value) {
      return value.f0;
    }
}
```

- Each selector is only returning the field on which the join should be performed
- Unfortunately, it's not possible to use lambda expressions here because Flink needs generic type info.

Join

- Next, let's implement merging logic using those selectors:

```
List<Tuple2<Tuple2<Integer, String>,
    Tuple3<Integer, String, String>>> joined
= transactions.join(addresses)
        .where(new IdKeySelectorTransaction())
        .equalTo(new IdKeySelectorAddress())
        .collect();
```

Sort

- Let’s say that you have the following collection of *Tuple2*:

```
Tuple2<Integer, String> secondPerson = new Tuple2<>(4, "Tom");
Tuple2<Integer, String> thirdPerson = new Tuple2<>(5, "Scott");
Tuple2<Integer, String> fourthPerson = new Tuple2<>(200, "Michael");
Tuple2<Integer, String> firstPerson = new Tuple2<>(1, "Jack");
DataSet<Tuple2<Integer, String>> transactions =
      env.fromElements( fourthPerson, secondPerson, thirdPerson,
firstPerson);
```

- If you want to sort this collection by the first field of the tuple, you can use the sortPartitions() transformation:

```
List<Tuple2<Integer, String>> sorted = transactions
      .sortPartition(new IdKeySelectorTransaction(), Order.ASCENDING)
      .collect();
```

Word Count

```
public class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
       @Override
       public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
              Stream.of(value.toLowerCase().split("\\W+"))
                     .filter(t -> t.length() > 0)
                     .forEach(token -> out.collect(new Tuple2<>(token, 1)));
       }
}

public static DataSet<Tuple2<String, Integer>> startWordCount(
       ExecutionEnvironment env, List<String> lines) throws Exception {
              DataSet<String> text = env.fromCollection(lines);
              return text.flatMap(new LineSplitter())
                            .groupBy(0)
                            .aggregate(Aggregations.SUM, 1);
}

List<String> lines = Arrays.asList(
       "This is a first sentence",
       "This is a second sentence with a one word");

DataSet<Tuple2<String, Integer>> result = WordCount.startWordCount(env, lines);

List<Tuple2<String, Integer>> collect = result.collect();
```

DataStream API

- Creating a DataStream
  - If we want to start consuming events, we first need to use the *StreamExecutionEnvironment* class:

```
StreamExecutionEnvironment executionEnvironment =
       StreamExecutionEnvironment.getExecutionEnvironment();
```

- We can create a stream of events using the ***executionEnvironment*** from a variety of sources
  - It could be some message bus like *Apache Kafka*, but in this example, we will simply create a source from a couple of string elements:

```
DataStream<String> dataStream = executionEnvironment.fromElements(
       "This is a first sentence",
       "This is a second sentence with a one word");
```

- We can apply transformations to every element of the *DataStream* like in the normal *DataSet* class:

```
SingleOutputStreamOperator<String> upperCase = text.map(String::toUpperCase);
```

- To trigger the execution, we need to invoke a sink operation such as *print()* that will just print the result of transformations to the standard output, followed with the *execute()* method on the *StreamExecutionEnvironment* class:

```
upperCase.print();
env.execute();
```

- It will produce the following output:

```
1> THIS IS A FIRST SENTENCE
2> THIS IS A SECOND SENTENCE WITH A ONE WORD
```

Windowing of Events

- When processing a stream of events in real-time, you may sometimes need to group events together and apply some computation on a window of those events
- Suppose we have a stream of events
  - each event is a pair consisting of the event number and the timestamp when the event was sent to our system
  - we can tolerate events that are out-of-order but only if they are no more than twenty seconds late
- Let’s first create a stream simulating two events that are several minutes apart and define a timestamp extractor that specifies our lateness threshold:

```
SingleOutputStreamOperator<Tuple2<Integer, Long>> windowed =
  env.fromElements(
  new Tuple2<>(16, ZonedDateTime.now().plusMinutes(25).toInstant().getEpochSecond()),
  new Tuple2<>(15, ZonedDateTime.now().plusMinutes(2).toInstant().getEpochSecond()))
  .assignTimestampsAndWatermarks(
      new BoundedOutOfOrdernessTimestampExtractor
            <Tuple2<Integer, Long>>(Time.seconds(20)) {
      @Override
      public long extractTimestamp(Tuple2<Integer, Long> element) {
            return element.f1 * 1000;
      }
  });
```

Windowing of Events

- Next, let's define a window operation to group our events into five-second windows and apply a transformation on those events:

```
SingleOutputStreamOperator<Tuple2<Integer, Long>> reduced = windowed
      .windowAll(TumblingEventTimeWindows.of(Time.seconds(5)))
      .maxBy(0, true);
reduced.print();
```

- It will get the last element of every five-second window, so it prints out:

```
1> (15,1491221519)
```

- Note that we do not see the second event because it arrived later than the specified lateness threshold.

Questions?

http://www.cs.ucy.ac.cy/~dzeina/courses/epl646/labs/lab.html