Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

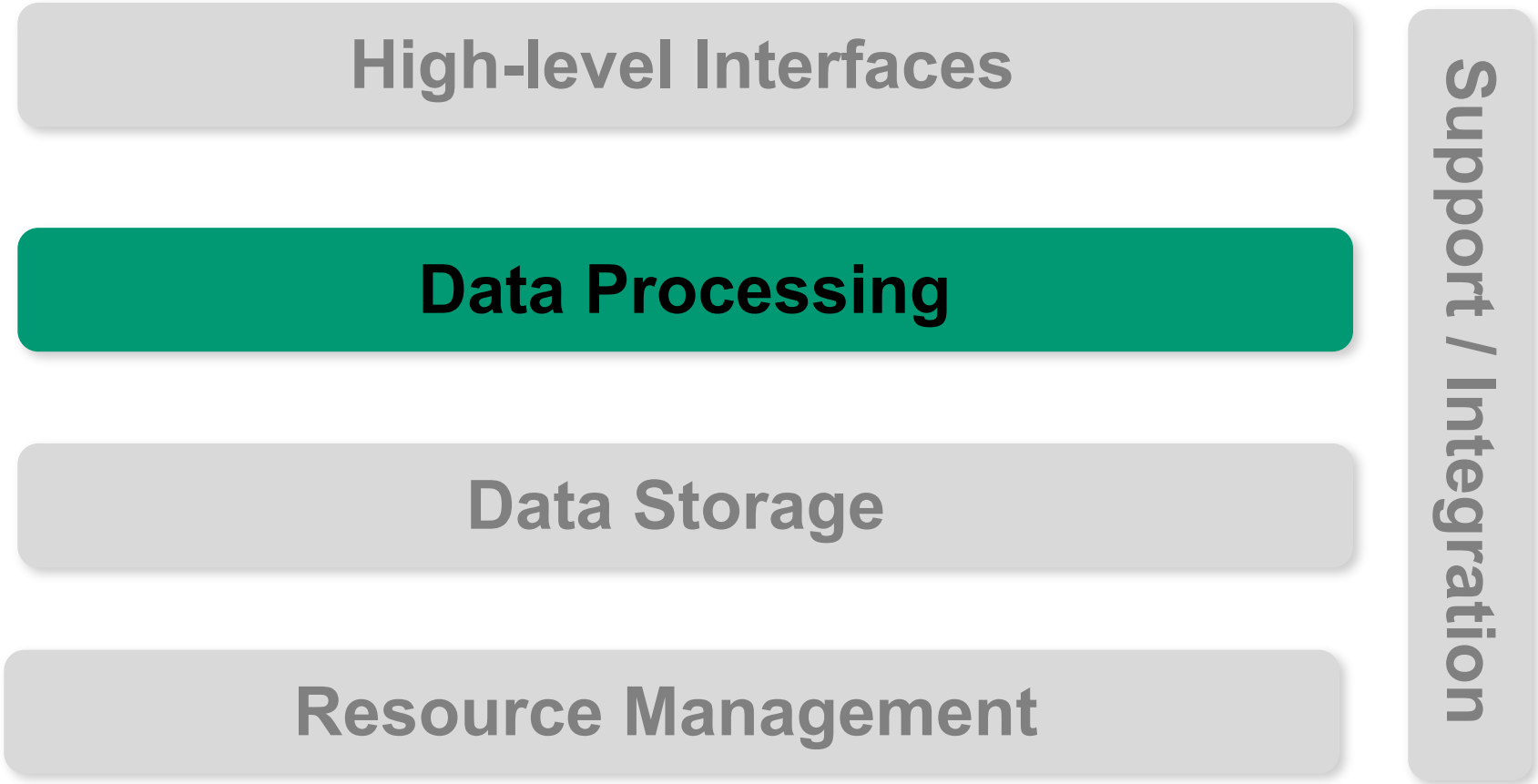
Apache Flink: Hands-on Session

A.A. 2022/23

Matteo Nardelli

Laurea Magistrale in Ingegneria Informatica - II anno

The reference Big Data stack



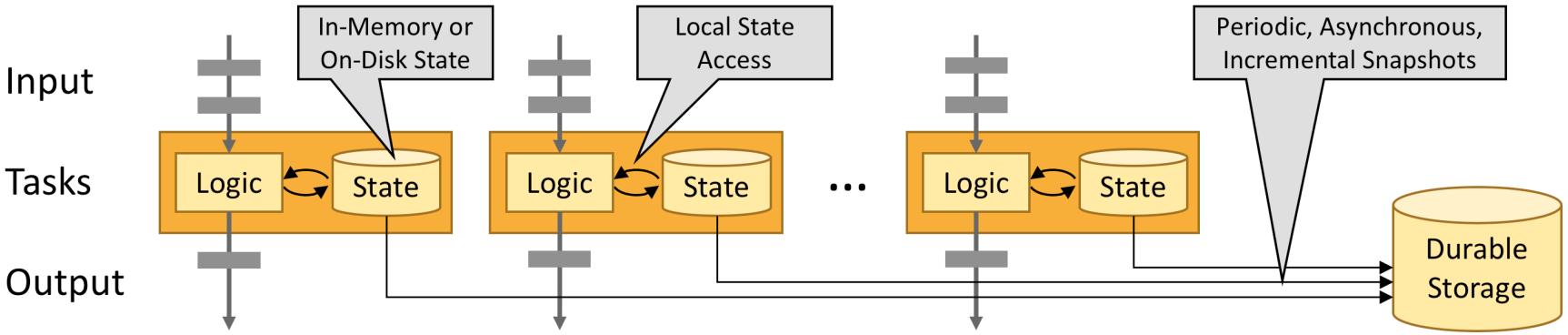
Apache Flink

- Apache Flink is a framework and distributed processing engine for stateful computations over *unbounded and bounded* data streams.
  - **Unbounded streams**: have a start but no defined end; must be continuously processed; is not possible to wait for all data to arrive.
    - Stream processing
  - **Bounded streams**: have defined start and end; can be processed by ingesting all data before computation; ordered ingestion is not usually required (can be sorted)
    - Batch processing

- Flink has been designed to run in *all common cluster environments*, perform computations at *in-memory speed* and at *any scale*.

Apache Flink

- Flink is designed to run stateful streaming applications at any scale.
  - Applications are parallelized into possibly thousands of tasks that are distributed and concurrently executed in a cluster.
- Leverage In-Memory Performance
  - Stateful Flink applications are optimized for local state access.

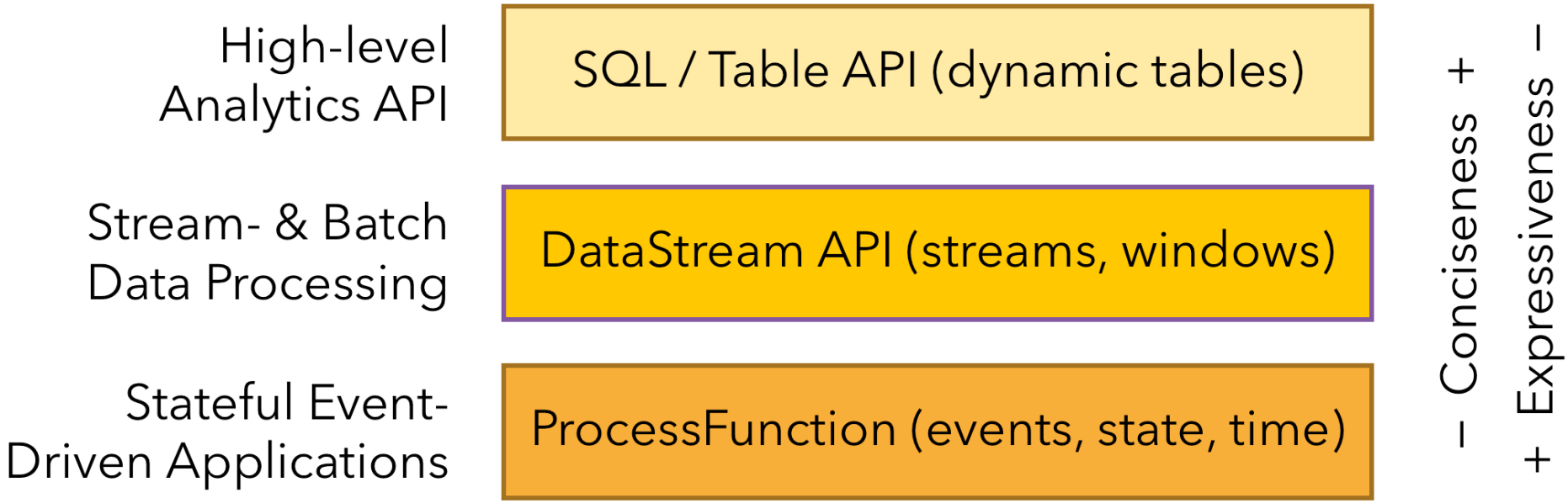


Apache Flink

- Key concepts:
  - Stream:
    - bounded/unbounded;
    - real-time/recorded
  - State:
    - Flink offers state primitives,
    - pluggable state backends (e.g., RocksDB),
    - exactly-once semantic,
    - scalable applications (data partitioning and distribution)
  - Time:
    - event-time vs processing-time mode;
    - watermark;
    - late data handling

Apache Flink: APIs

- Multiple APIs at different levels of abstraction



Apache Flink: ProcessFunction API

ProcessFunction API:

- Low-level stream processing operation
- Handles events by being invoked for each event received
- Has access to (RuntimeContext):
  - Events (stream elements)
  - State (fault-tolerant, consistent, only on keyed stream)
  - Timers (event time and processing time, only on keyed stream)

Read more: https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/operators/process_function/

Apache Flink: DataStream API

- Data streaming applications: **DataStream API**
  - Supports functional transformations on data streams, with user-defined state and flexible windows
  - Example: windowed version of WordCount

WindowWordCount using Flink's DataStream API

Sliding time window of 10 sec length and 5 sec slide

```
final int windowSize = params.getInt("window", 10);
final int slideSize = params.getInt("slide", 5);

DataStream<Tuple2<String, Integer>> counts =
// split up the lines in pairs (2-tuples) containing: (word,1)
text.flatMap(new WordCount.Tokenizer())
        // create windows of windowSize records slided every slideSize records
        .keyBy(0)
        .countWindow(windowSize, slideSize)
        // group by the tuple field "0" and sum up tuple field "1"
        .sum(1);
```

See https://bit.ly/2AhCEBX

Apache Flink: DataStream API

DataStream API:

- Provides primitives for many common stream processing operations:
  - Windowing
  - Record-at-a-time transformations
  - Enriching events
- Based on functions, e.g., map(), reduce(), and aggregate()

```
DataStream<Tuple2<String, Long>> result = words
        .map(word -> Tuple2.of(word, 1L))
        .returns(Types.TUPLE(Types.STRING, Types.LONG))
        .keyBy(0)
        .reduce((a, b) -> Tuple2.of(a.f0, a.f1 + b.f1));
```

Apache Flink: Table API and SQL

Table API

- Table API and SQL are unified APIs for batch and stream processing;
- They can be seamlessly integrated with the DataStream and DataSet APIs;
- They support user-defined scalar, aggregate, and table-valued functions.
- Relational APIs are designed to ease the definition of data analytics, data pipelining, and ETL applications

Sessionize a clickstream and count the number of clicks per session

```
SELECT userId, COUNT(*)
FROM clicks
GROUP BY SESSION(clicktime, INTERVAL '30' MINUTE), userId
```

Flink: APIs and libraries

- Batch processing applications: **DataSet API**
  - Supports a wide range of data types beyond key/value pairs and a wealth of operators

Core of PageRank algorithm using DataSet API

```
DataSet<Tuple2<Long, Double>> newRanks = iteration
        // join pages with outgoing edges and distribute rank
        .join(adjacencyListInput).where(0).equalTo(0).flatMap(new JoinVertexWithEdgesMatch())
        // collect and sum ranks
        .groupBy(0).aggregate(SUM, 1)
        // apply dampening factor
        .map(new Dampener(DAMPENING_FACTOR, numPages));

DataSet<Tuple2<Long, Double>> finalPageRanks = iteration.closeWith(
        newRanks,
        newRanks.join(iteration).where(0).equalTo(0)
        // termination condition
        .filter(new EpsilonFilter()));
```

See https://bit.ly/2zEH3Pk

Anatomy of a Flink program

- Let's analyze DataStream API

https://ci.apache.org/projects/flink/flink-docs-stable/dev/datastream_api.html

- Special `DataStream` class used to represent a collection of data in a Flink program
- Each Flink program consists of the same basic parts:
  1. Obtain one execution environment

```
getExecutionEnvironment()

createLocalEnvironment()

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

  2. Load/create initial data

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

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

Anatomy of a Flink program

3. Specify transformations on data by calling methods on `DataStream`

```
DataStream<String> input = ...;

DataStream<Integer> parsed = input.map(new MapFunction<String, Integer>() {
    @Override
    public Integer map(String value) {
        return Integer.parseInt(value);
    }
});
```

4. Specify where to put the results of your computations

```
writeAsText(String path)

print()
```

5. Trigger the program execution by calling `execute` on `StreamExecutionEnvironment`

Flink: Lazy evaluation

- Flink programs are executed lazily
  - When program's main method is executed, data loading and transformations do not happen directly
  - Rather, each operation is created and added to program's plan
  - Operations are actually executed when execution is explicitly triggered by calling `execute()` on the execution environment

Flink: data sources

- Several predefined stream sources accessible from the StreamExecutionEnvironment

1. File-based:
   - E.g., `readTextFile(path)` to read text files
   - Flink splits file reading process into two sub-tasks: *directory monitoring* and *data reading*
     - Monitoring is implemented by a single, non-parallel task, while reading is performed by multiple tasks running in parallel, whose parallelism is equal to the job parallelism
2. Socket-based
3. Collection-based
4. Custom
   - E.g., to read from Kafka `fromSource(new KafkaSource<…>(…))` https://nightlies.apache.org/flink/flink-docs-stable/docs/connectors/datastream/kafka/
   - See Apache Bahir for streaming connectors and SQL data sources https://bahir.apache.org/

Flink: DataStream transformations

- **Map**
  DataStream → DataStream
  - Example: double 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
  - Example: split 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);
        }
    }
});
```

Flink: DataStream transformations

- **Filter**
  DataStream → DataStream
  - Example: filter out zero values

```
dataStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 0;
    }
});
```

- **KeyBy**
  DataStream → KeyedStream
  - To specify a key that logically partitions a stream into disjoint partitions
  - Internally, implemented with hash partitioning
  - Different ways to specify keys, the simplest case is grouping tuples on one or more fields of the tuple
  - Examples:

```
dataStream.keyBy("someKey") // Key by field "someKey"
dataStream.keyBy(0) // Key by the first element of a Tuple
```

Flink: DataStream transformations

- **Reduce**
  KeyedStream → DataStream
  - "Rolling" reduce on a keyed data stream
  - Combines the current element with the last reduced value and emits the new value
  - Example: create a stream of partial sums

```
keyedStream.reduce(new ReduceFunction<Integer>() {
    @Override
    public Integer reduce(Integer value1, Integer value2)
    throws Exception {
        return value1 + value2;
    }
});
```

Flink: DataStream transformations

- **Aggregations**
  KeyedStream → DataStream
  - To aggregate on a keyed data stream
  - min returns the minimum value, whereas minBy returns the element that has the minimum value in this field

```
keyedStream.sum(0);
keyedStream.sum("key");
keyedStream.min(0);
keyedStream.min("key");
keyedStream.max(0);
keyedStream.max("key");
keyedStream.minBy(0);
keyedStream.minBy("key");
```

- **Window**
  KeyedStream → WindowedStream

```
dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5))); // Last
5 seconds of data
```

Flink: DataStream transformations

- Other transformations available in Flink
  - **join:** joins two data streams on a given key
  - **union:** union of two or more data streams creating a new stream containing all the elements from all the streams
  - **split**: splits the stream into two or more streams according to some criterion
  - **iterate**: creates a “feedback” loop in the flow, by redirecting the output of one operator to some previous operator
    - Useful for algorithms that continuously update a model

See https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/operators/overview/

Example: streaming window WordCount

- Count the words from a web socket in 5 sec windows

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class WindowWordCount {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Tuple2<String, Integer>> dataStream = env
                .socketTextStream("localhost", 9999)
                .flatMap(new Splitter())
                .keyBy(0)          // Key by the first element of a Tuple
                .timeWindow(Time.seconds(5))
                .sum(1);

        dataStream.print();

        env.execute("Window WordCount");
    }
```

Example: streaming window WordCount

```
    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));
            }
        }
    }

}
```

Flink: windows support

- Windows can be applied either to *keyed* streams or to *non-keyed* ones
- General structure of a windowed Flink program

Keyed Windows

```
stream
       .keyBy(...)               <-  keyed versus non-keyed windows
       .window(...)              <-  required: "assigner"
      [.trigger(...)]            <-  optional: "trigger" (else default trigger)
      [.evictor(...)]            <-  optional: "evictor" (else no evictor)
      [.allowedLateness(...)]    <-  optional: "lateness" (else zero)
      [.sideOutputLateData(...)] <-  optional: "output tag" (else no side output for late data)
       .reduce/aggregate/fold/apply()      <-  required: "function"
      [.getSideOutput(...)]      <-  optional: "output tag"
```

Non-Keyed Windows

```
stream
       .windowAll(...)           <-  required: "assigner"
      [.trigger(...)]            <-  optional: "trigger" (else default trigger)
      [.evictor(...)]            <-  optional: "evictor" (else no evictor)
      [.allowedLateness(...)]    <-  optional: "lateness" (else zero)
      [.sideOutputLateData(...)] <-  optional: "output tag" (else no side output for late data)
       .reduce/aggregate/fold/apply()      <-  required: "function"
      [.getSideOutput(...)]      <-  optional: "output tag"
```

Flink: window lifecycle

- First, specify if stream is keyed or not and define the *window assigner*
  - Keyed stream allows to perform the windowed computation in parallel by multiple tasks
  - The window is completely removed when the time (event or processing time) passes its end timestamp plus the user-specified *allowed lateness*

- Then, associate to window its *trigger, (evictor)* and *function*
  - Trigger determines when a window is ready to be processed by the window function
  - Evictor (optional) has the ability to remove elements from a window after the trigger fires and before and/or after the window function is applied
  - Function specifies the computation to be applied to the window contents

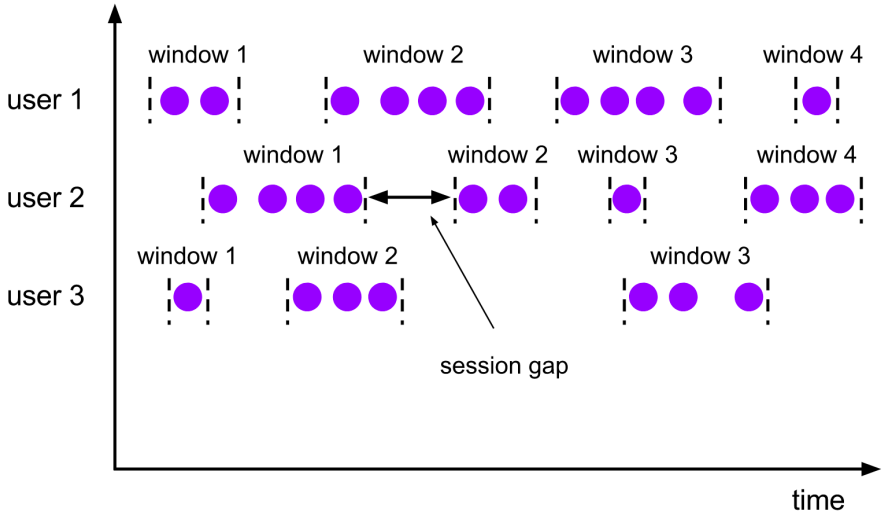
Read more: https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/operators/windows/

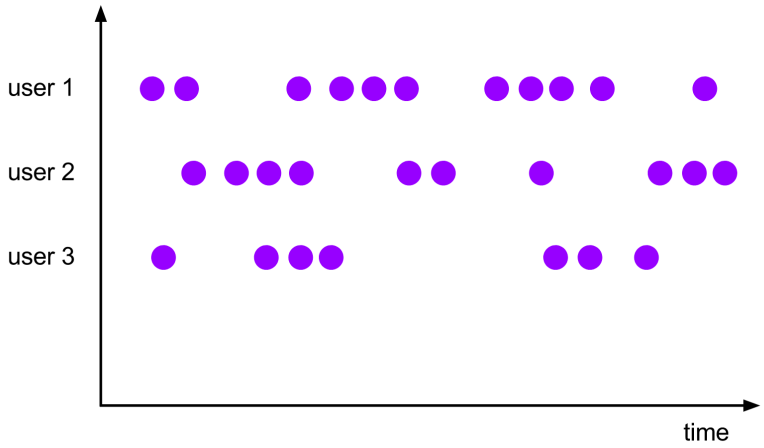
Flink: window assigners

- How elements are assigned to windows
- Support for different window assigners
  - Each WindowAssigner comes with a default Trigger
- Built-in assigners for most common use cases:
  - Tumbling windows
  - Sliding windows
  - Session windows
  - Global windows
- Except for global windows, they assign elements to windows based on time, which can either be processing time or event time
- It is also possible to implement a custom window assigner

Flink: window assigners

- Session windows
  - To group elements by sessions of activity
  - Differently from tumbling and sliding windows, do not overlap and do not have a fixed start and end time
  - A session window closes when a gap of inactivity occurs
- Global windows
  - To assign all elements with the same key to the same single global window
  - Only useful if you also specify a custom trigger





Flink: window functions

- Different window functions to specify the computation on each window

- **ReduceFunction**
  - To incrementally aggregate the elements of a window
  - Example: sum up the second fields of the tuples for all elements in a window

```
DataStream<Tuple2<String, Long>> input = ...;

input
    .keyBy(<key selector>)
    .window(<window assigner>)
    .reduce(new ReduceFunction<Tuple2<String, Long>> {
      public Tuple2<String, Long> reduce(Tuple2<String, Long> v1, Tuple2<String, Long> v2) {
        return new Tuple2<>(v1.f0, v1.f1 + v2.f1);
      }
    });
```

Flink: window functions

- **AggregateFunction**: generalized version of a ReduceFunction
  - Example: compute average of the elements in the window

```
// the accumulator, which holds the state of the in-flight aggregate
public class AverageAccumulator {
    long count;
    long sum;
}

// implementation of an aggregation function for an 'average'
public class Average implements AggregateFunction<Integer, AverageAccumulator, Double> {

    public AverageAccumulator createAccumulator() {
        return new AverageAccumulator();
    }

    public AverageAccumulator merge(AverageAccumulator a, AverageAccumulator b) {
        a.count += b.count;
        a.sum += b.sum;
        return a;
    }

    public void add(Integer value, AverageAccumulator acc) {
        acc.sum += value;
        acc.count++;
    }

    public Double getResult(AverageAccumulator acc) {
        return acc.sum / (double) acc.count;
    }
}
```

Flink: window functions

- **AggregateFunction**
  - Example: compute weighted average of the elements in the window

```
// implementation of a weighted average
// this reuses the same accumulator type as the aggregate function for 'average'
public class WeightedAverage implements AggregateFunction<Datum, AverageAccumulator, Double> {

    public AverageAccumulator createAccumulator() {
        return new AverageAccumulator();
    }

    public AverageAccumulator merge(AverageAccumulator a, AverageAccumulator b) {
        a.count += b.count;
        a.sum += b.sum;
        return a;
    }

    public void add(Datum value, AverageAccumulator acc) {
        acc.count += value.getWeight();
        acc.sum += value.getValue();
    }

    public Double getResult(AverageAccumulator acc) {
        return acc.sum / (double) acc.count;
    }
}
```

Flink: window functions

- **ProcessWindowFunction**: gets an Iterable containing all the elements of the window, and a Context object with access to time and state information
  - ✓ More flexibility than other window functions
  - ✗ At the cost of performance and resource consumption: elements are buffered until the window is ready for processing

```
input
  .keyBy(t -> t.f0)
  .window(TumblingEventTimeWindows.of(Time.minutes(5)))
  .process(new MyProcessWindowFunction());
```

```
public class MyProcessWindowFunction
    extends ProcessWindowFunction<Tuple2<String, Long>, String, String, Ti

  @Override
  public void process(String key, Context context, Iterable<Tuple2<String,
    long count = 0;
    for (Tuple2<String, Long> in: input) {
      count++;
    }
    out.collect("Window: " + context.window() + "count: " + count);
  }
}
```

Flink: window functions

- **ProcessWindowFunction**: gets an Iterable containing all the elements of the window, and a Context object with access to time and state information
  - ✓ More flexibility than other window functions
  - ✗ At the cost of performance and resource consumption: elements are buffered until the window is ready for processing

- ReduceFunction and AggregateFunction can execute more efficiently
  - Flink can incrementally aggregate the elements for each window as they arrive

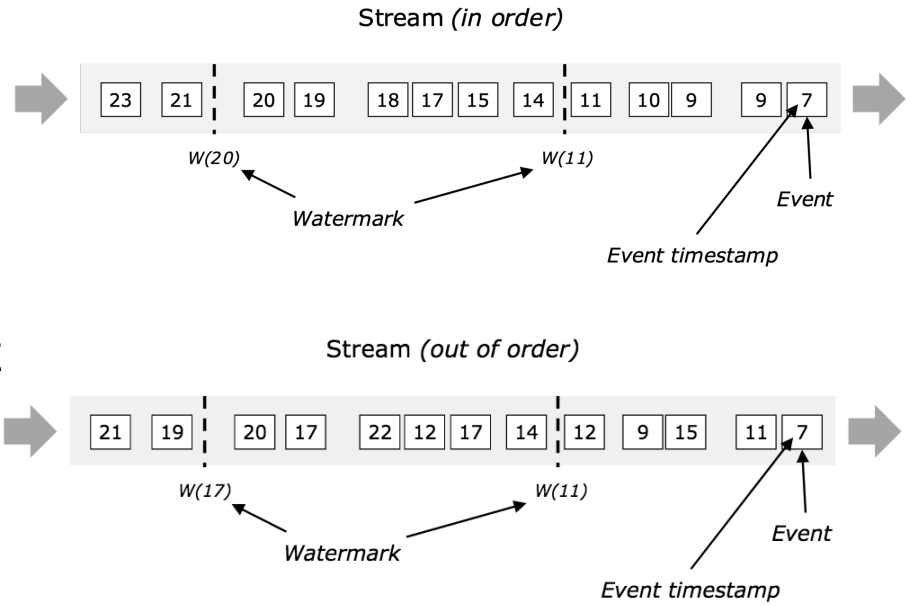
Flink: control events

- Control events: special events injected in the data stream by operators

- Two types of control events in Flink
  - Watermarks
  - Checkpoint barriers

Flink: watermarks

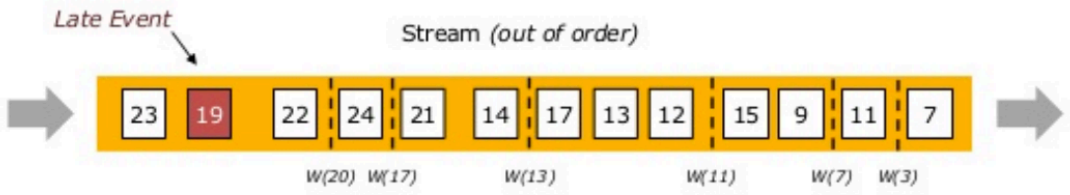
- Watermarks mark the progress of event time within a data stream
- Flow as part of data stream and carry a timestamp *t*
  - *W(t)* declares that event time has reached time *t* in that stream, meaning that there should be no more elements with timestamp *t'* <= *t*
  - Crucial for *out-of-order* streams, where events are not ordered by their timestamps



Read more: https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/event-time/generating_watermarks/

Flink: watermarks

- By default, late elements are dropped when the watermark is past the end of the window
- However, Flink allows to specify a maximum *allowed lateness* for window operator
  - By how much time elements can be late before they are dropped (0 by default)
  - Late elements that arrive after the watermark has passed the end of the window but before it passes the end of the window plus the allowed lateness, are still added to the window

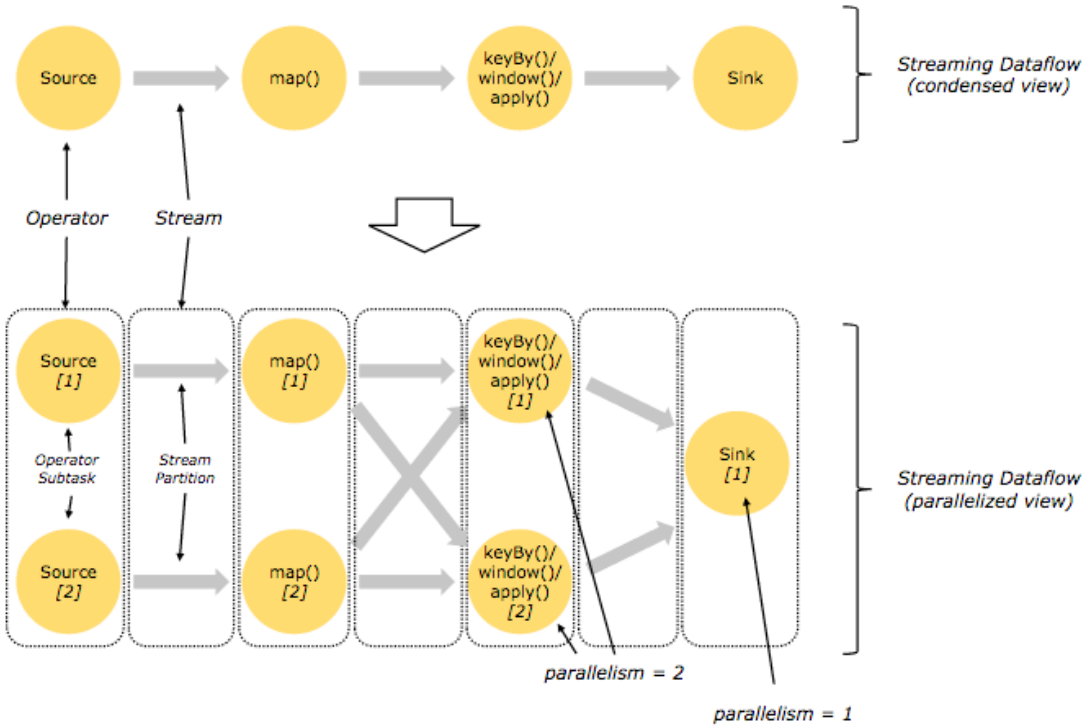


Flink: watermarks

- Flink does not provide ordering guarantees after any form of stream partitioning or broadcasting
  - In such case, dealing with out-of-order tuples is left to the operator implementation

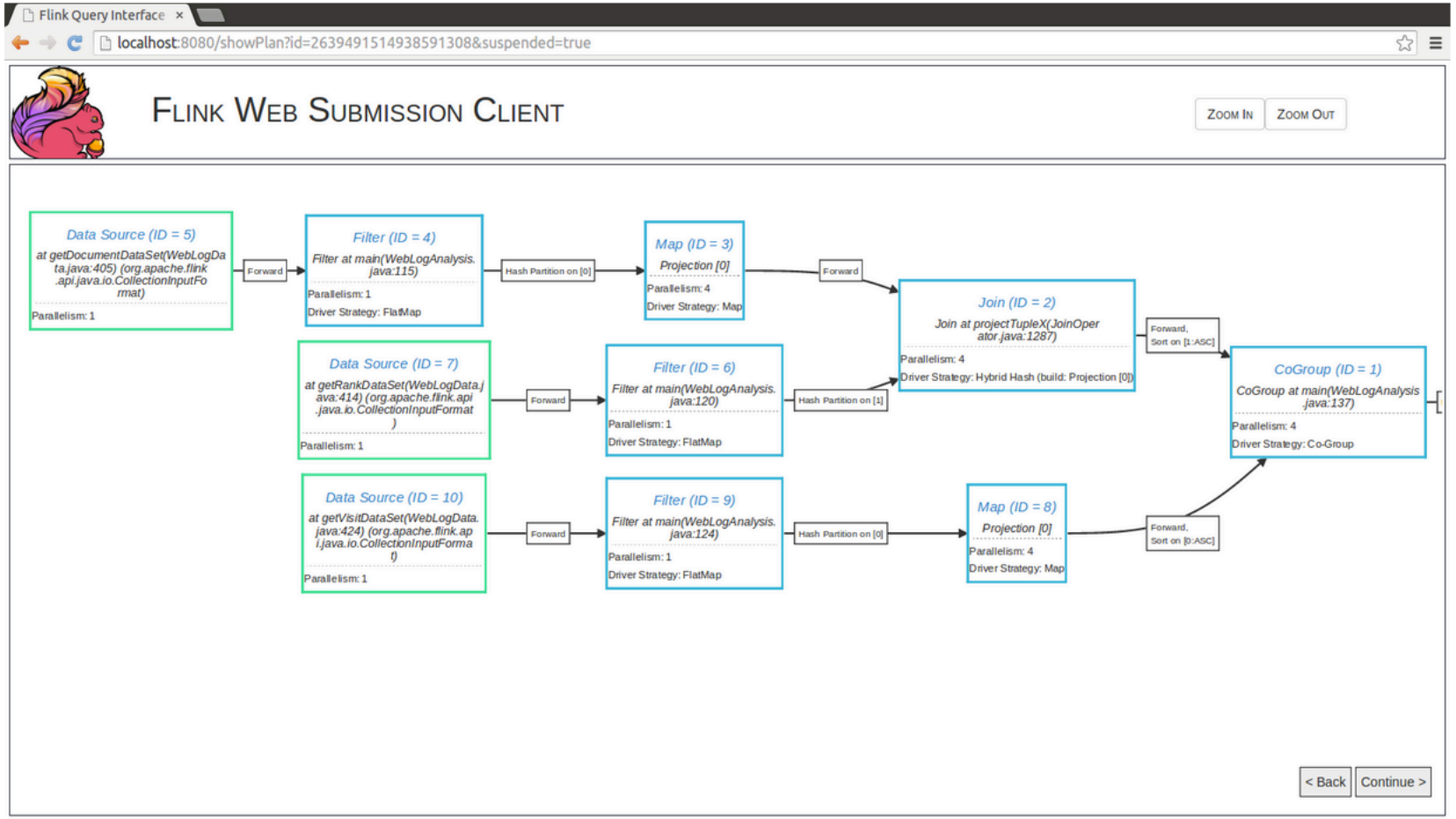
Flink: application execution

- Data parallelism
  - Different operators of the same program may have different levels of parallelism
  - The parallelism of an individual operator, data source, or data sink can be defined by calling its `setParallelism()` method



Flink: application execution

- Execution plan can be visualized



Flink: application monitoring

- Built-in monitoring and metrics system
- Allows gathering and exposing metrics to external systems
- Built-in metrics include
  - Throughput: in terms of number of records per sec. (per operator/ task)
  - Latency
    - Support for latency tracking: special markers (called LatencyMarker) are periodically inserted at all sources in order to obtain a distribution of latency between sources and each downstream operator
      - But do not account for time spent in operator processing (or in window buffers)
      - Assume that all machines clocks are sync
  - Used JVM heap/non-heap/direct memory
  - Availability, checkpointing

Flink: application monitoring

- Application-specific metrics can be added
  - E.g., counters for number of invalid records
- All metrics can be
  - queried via Flink's Monitoring REST API
  - visualized in Flink's Dashboard (Metrics tab)
  - or send to external systems (e.g., Graphite and InfluxDB)

See https://ci.apache.org/projects/flink/flink-docs-stable/monitoring/metrics.html