EIT Summer School 2019

Apache Flink

Based on https://training.ververica.com

Maximilian Michels <mxm@apache.org>
Software Engineer / Consultant
Committer @ Apache Beam / Apache Flink

@stadtlegende

Dr Paris Carbone <paris.carbone@ri.se>
Senior Researcher @ RISE
Committer @ Apache Flink

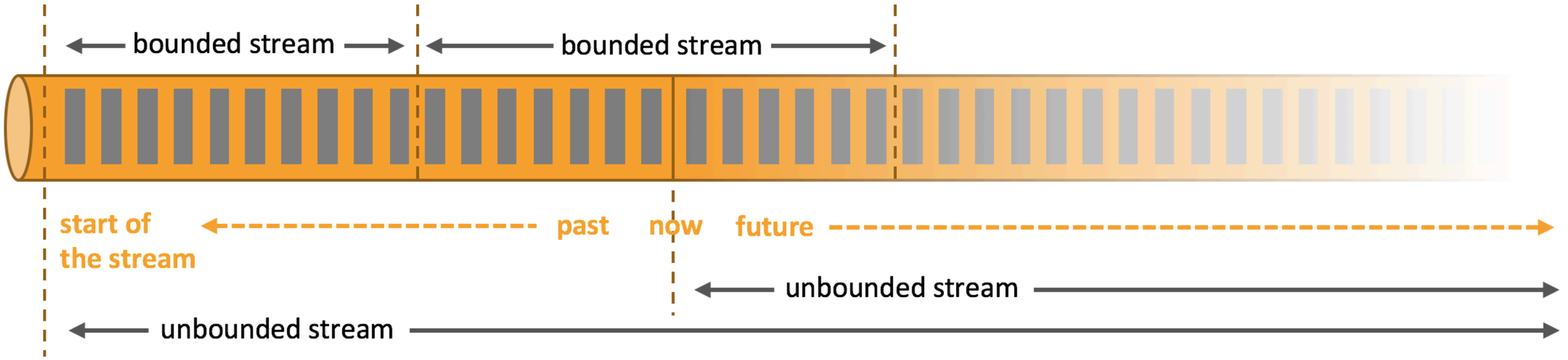
@SenorCarbone

Contents

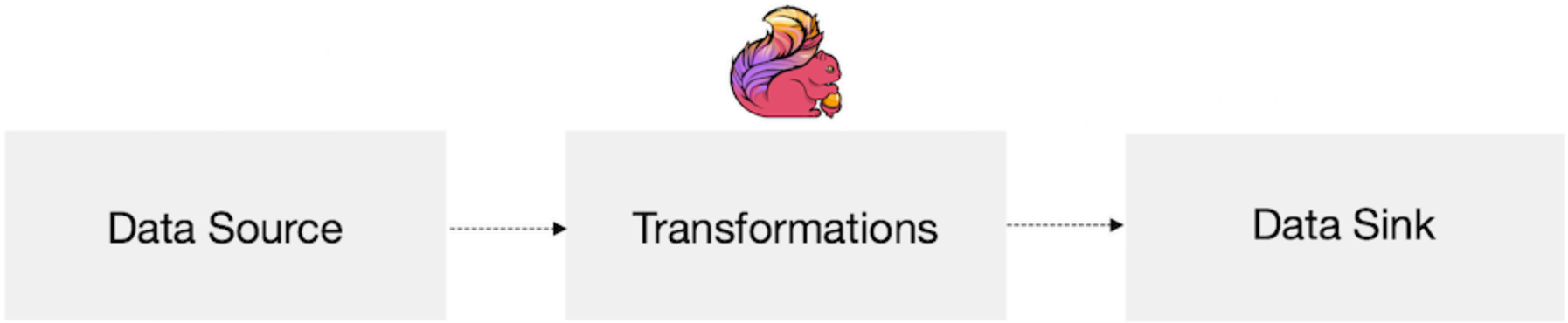
- DataSet API
- **DataStream API**
  - Concepts
  - Set up an environment to develop Flink programs
  - Implement streaming data processing pipelines
    - Flink managed state
    - Event time

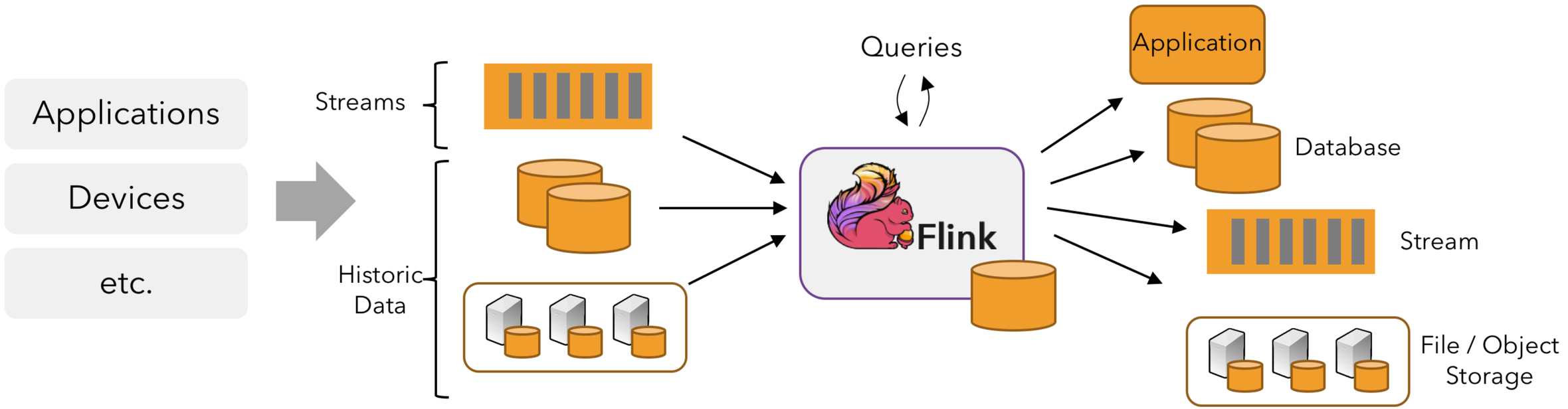
Streaming in Apache Flink

- Streams are natural
  - Events of any type like sensors, click streams, logs
- Batch processing as a subset of stream processing

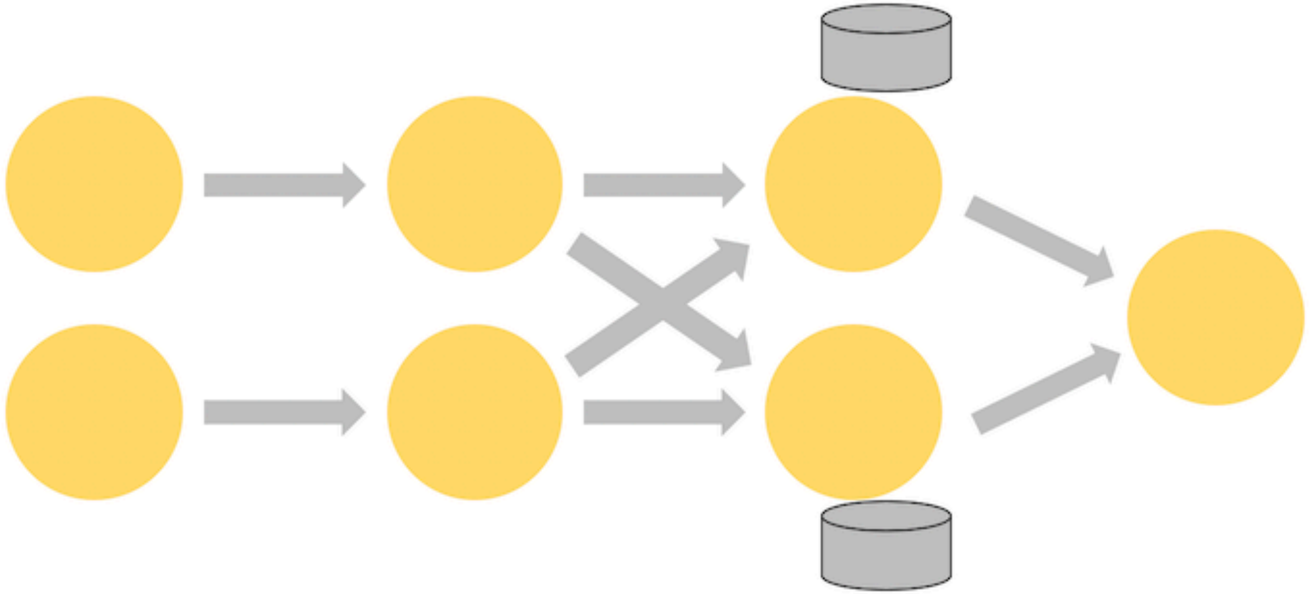


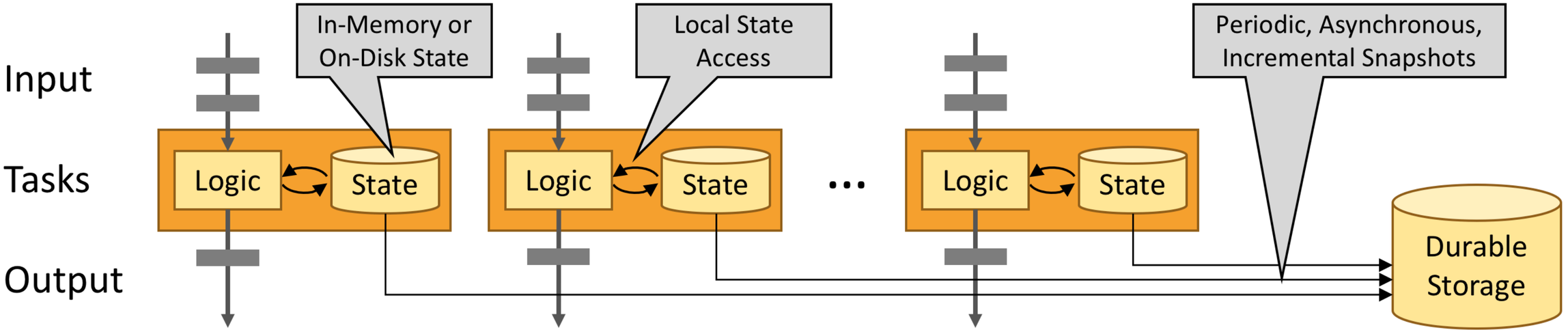
Processing Data

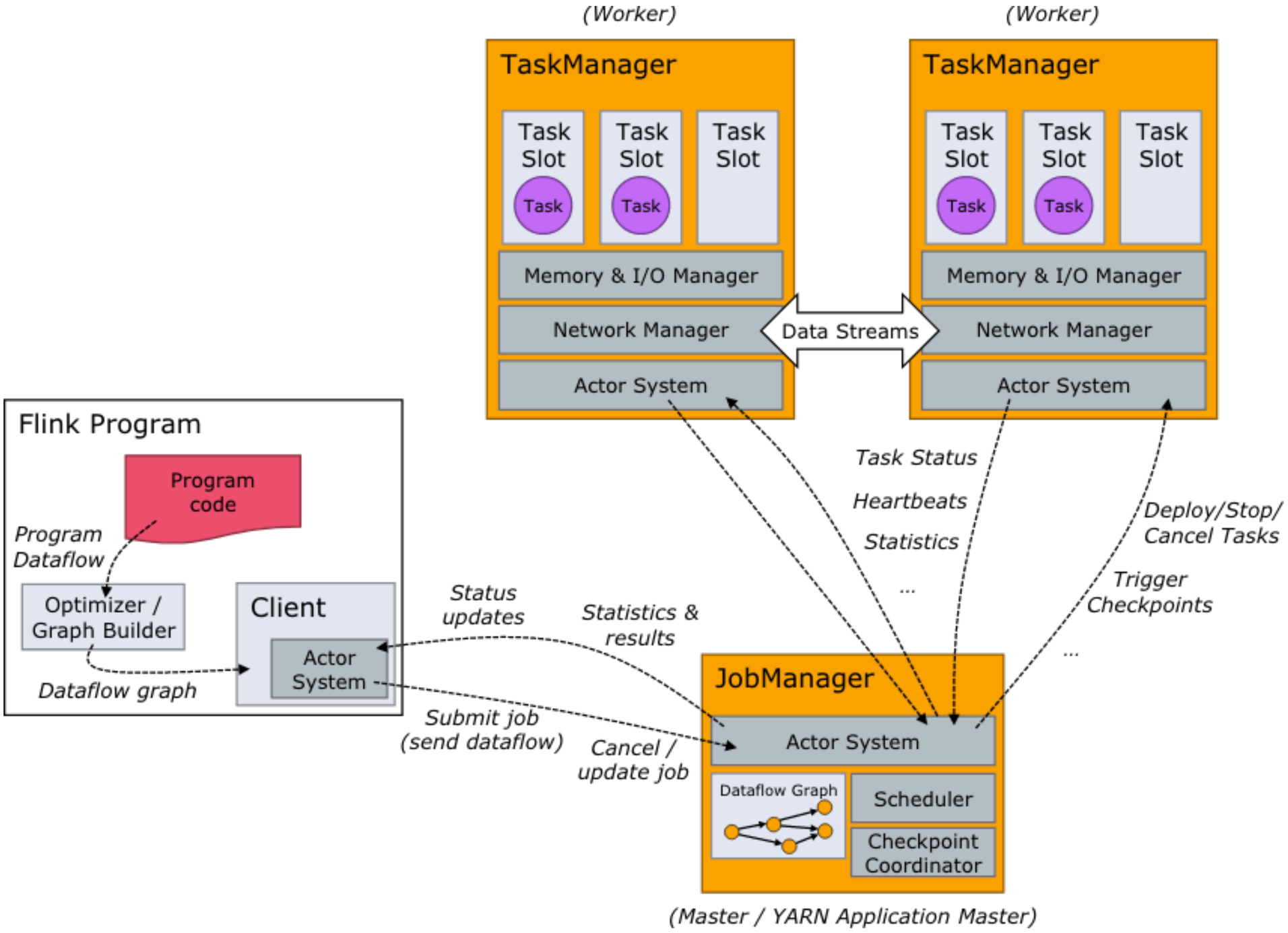




Dataflows




Input
Tasks
Output
In-Memory or On-Disk State
Local State Access
Periodic, Asynchronous, Incremental Snapshots
Logic
State
Logic
State
...
Logic
State
Durable Storage


(Worker)
(Worker)
TaskManager
TaskManager
Task Slot
Task Slot
Task Slot
Task
Task
Memory & I/O Manager
Network Manager
Actor System
Data Streams
Flink Program
Program code
Program Dataflow
Optimizer / Graph Builder
Client
Actor System
Dataflow graph
Status updates
Statistics & results
Submit job (send dataflow)
Cancel / update job
Task Status
Heartbeats
Statistics
...
Deploy/Stop/ Cancel Tasks
Trigger Checkpoints
...
JobManager
Actor System
Dataflow Graph
Scheduler
Checkpoint Coordinator
(Master / YARN Application Master)

Let's Talk About Time

- **Processing Time**
- **Event Time**
- Events may arrive **out of order**!

What Can Be Streamed?

- Anything (if you write a serializer/deserializer for it)
- Flink has a built-in type system which supports:
  - basic types, i.e., String, Long, Integer, Boolean, Array
  - composite types: Tuples, POJOs, and Scala case classes
  - Kryo for unknown types

Type Examples

Tuples

Tuple1 through Tuple25 types.

```
Tuple2<String, Integer> person =
    new Tuple2<>("Fred", 35);

// zero based index!
String name = person.f0;
Integer age = person.f1;
```

POJOs

A POJO (plain old Java object) is any Java class that

- has an empty default constructor
- all fields are either
  - public, or
  - have a default getter and setter

```
public class Person {
    public String name;
    public Integer age;
    public Person() {};
    public Person(String name, Integer age) {
        …
    };
}

Person person = new Person("Fred Flintstone", 35);
```

Setup

- https://training.ververica.com/devEnvSetup.html
- Datasets:
  wget http://training.ververica.com/trainingData/nycTaxiRides.gz
  wget http://training.ververica.com/trainingData/nycTaxiFares.gz
- Walkthrough an example

Taxi Rides Dataset

Taxi Ride Events

rideId	Long	a unique id for each ride
taxiId	Long	a unique id for each taxi
driverId	Long	a unique id for each driver
isStart	Boolean	TRUE for ride start events, FALSE for ride end events
startTime	DateTime	the start time of a ride
endTime	DateTime	the end time of a ride, ""1970-01-01 00:00" for start events
startLon	Float	the longitude of the ride start location
startLat	Float	the latitude of the ride start location
endLon	Float	the longitude of the ride end location
endLat	Float	the latitude of the ride end location
passengerCnt	Short	number of passengers on the ride

Taxi Fare Dataset

Taxi Fare Events

rideId	Long	a unique id for each ride
taxiId	Long	a unique id for each taxi
driverId	Long	a unique id for each driver
startTime	DateTime	the start time of a ride
paymentType	String	CSH or CRD
tip	Float	tip for this ride
tolls	Float	tolls for this ride
totalFare	Float	total fare collected

Lab 1 -- Ride Cleansing

Transforming Data

Transforming Data

```
public static class EnrichedRide extends TaxiRide {
  public int startCell;
  public int endCell;

  public EnrichedRide() {}

  public EnrichedRide(TaxiRide ride) {
    this.rideId = ride.rideId;
    this.isStart = ride.isStart;
    ...
    this.startCell = GeoUtils.mapToGridCell(ride.startLon, ride.startLat);
    this.endCell = GeoUtils.mapToGridCell(ride.endLon, ride.endLat);
  }

  public String toString() {
    return super.toString() + "," +
      Integer.toString(this.startCell) + "," +
      Integer.toString(this.endCell);
  }
}
```

Map Function

```
public static class Enrichment implements MapFunction<TaxiRide, EnrichedRide> {
  @Override
  public EnrichedRide map(TaxiRide taxiRide) throws Exception {
    return new EnrichedRide(taxiRide);
  }
}
```

```
DataStream<TaxiRide> rides = env.addSource(new TaxiRideSource(...));

DataStream<EnrichedRide> enrichedNYCRides = rides
    .filter(new RideCleansing.NYCFilter())
    .map(new Enrichment());

enrichedNYCRides.print();
```

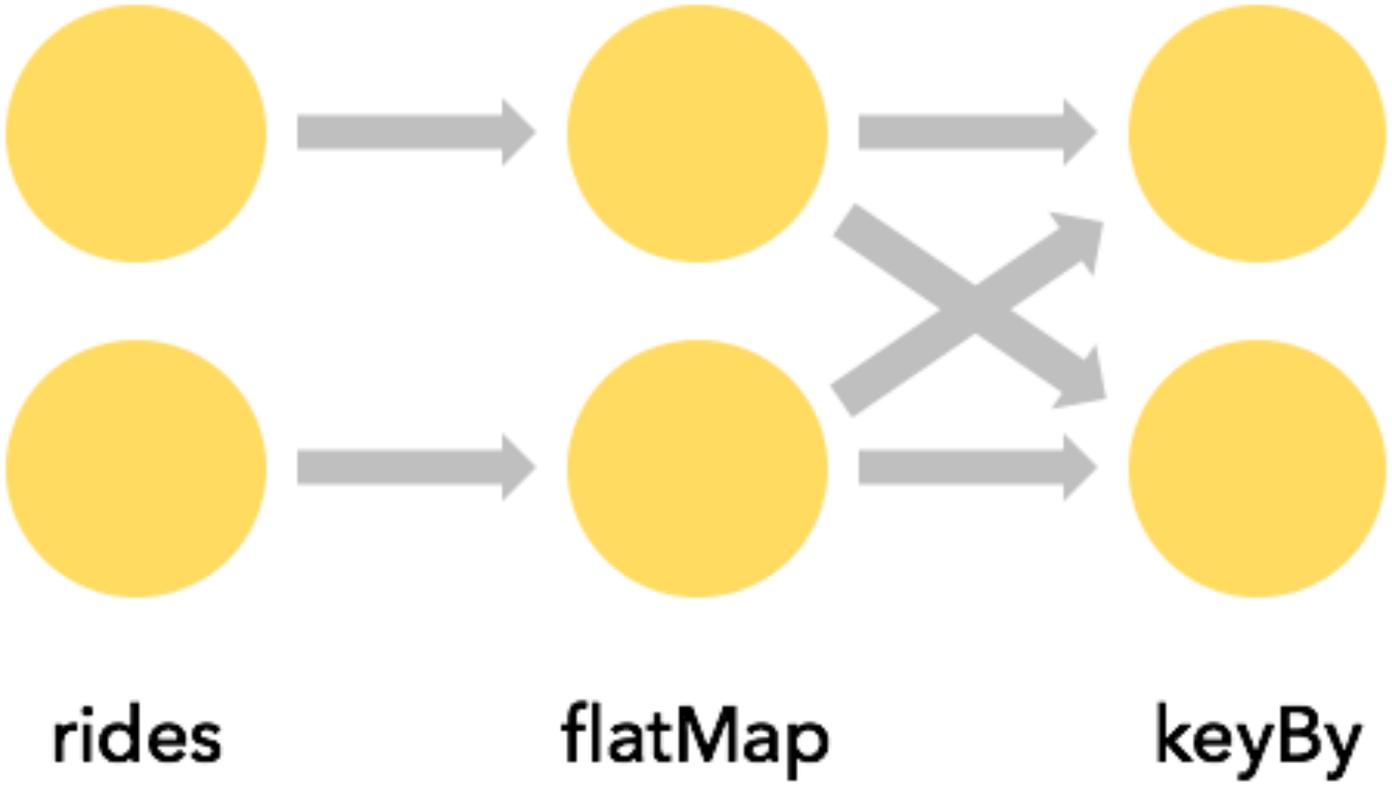
FlatMap Function

```
public static class NYCEnrichment implements FlatMapFunction<TaxiRide, EnrichedRide> {
  @Override
  public void flatMap(TaxiRide taxiRide, Collector<EnrichedRide> out) throws Exception {
    FilterFunction<TaxiRide> valid = new RideCleansing.NYCFilter();
    if (valid.filter(taxiRide)) {
      out.collect(new EnrichedRide(taxiRide));
    }
  }
}
```

```
DataStream<TaxiRide> rides = env.addSource(new TaxiRideSource(...));

DataStream<EnrichedRide> enrichedNYCRides = rides
    .flatMap(new NYCEnrichment());

enrichedNYCRides.print();
```

Keyed Streams

```
rides
  .flatMap(new NYCEnrichment())
  .keyBy("startCell")
```



KeyBy

```
rides
  .flatMap(new NYCEnrichment())
  .keyBy(
    new KeySelector<EnrichedRide, int>() {
      @Override
      public int getKey(EnrichedRide ride) throws Exception {
        return ride.startCell;
      }
    })
```

```
rides
  .flatMap(new NYCEnrichment())
  .keyBy(ride -> ride.startCell)
```

Keyed Aggregations

```
DataStream<Tuple2<Integer, Minutes>> minutesByStartCell =
  enrichedNYCRides
    .flatMap(new FlatMapFunction<EnrichedRide, Tuple2<Integer, Minutes>>() {
      @Override
      public void flatMap(EnrichedRide ride,
                          Collector<Tuple2<Integer, Minutes>> out) throws Exception {
        if (!ride.isStart) {
          Interval rideInterval = new Interval(ride.startTime, ride.endTime);
          Minutes duration = rideInterval.toDuration().toStandardMinutes();
          out.collect(new Tuple2<>(ride.startCell, duration));
        }
      }
    });
```

```
minutesByStartCell
    .keyBy(0) // startCell
    .maxBy(1) // duration
    .print();
```

```
...
4> (64549,5M)
4> (46298,18M)
1> (51549,14M)
1> (53043,13M)
1> (56031,22M)
1> (50797,6M)
...
1> (50797,8M)
...
1> (50797,11M)
...
1> (50797,12M)
```

Stateful Transformations

- local: Flink state is kept local to the machine that processes it
- durable: Flink state is automatically checkpointed and restored
- vertically scalable: Flink state can be kept in embedded RocksDB instances that scale by adding more local disk
- horizontally scalable: Flink state is redistributed as your cluster grows and shrinks
- queryable: Flink state can be queried via a REST API

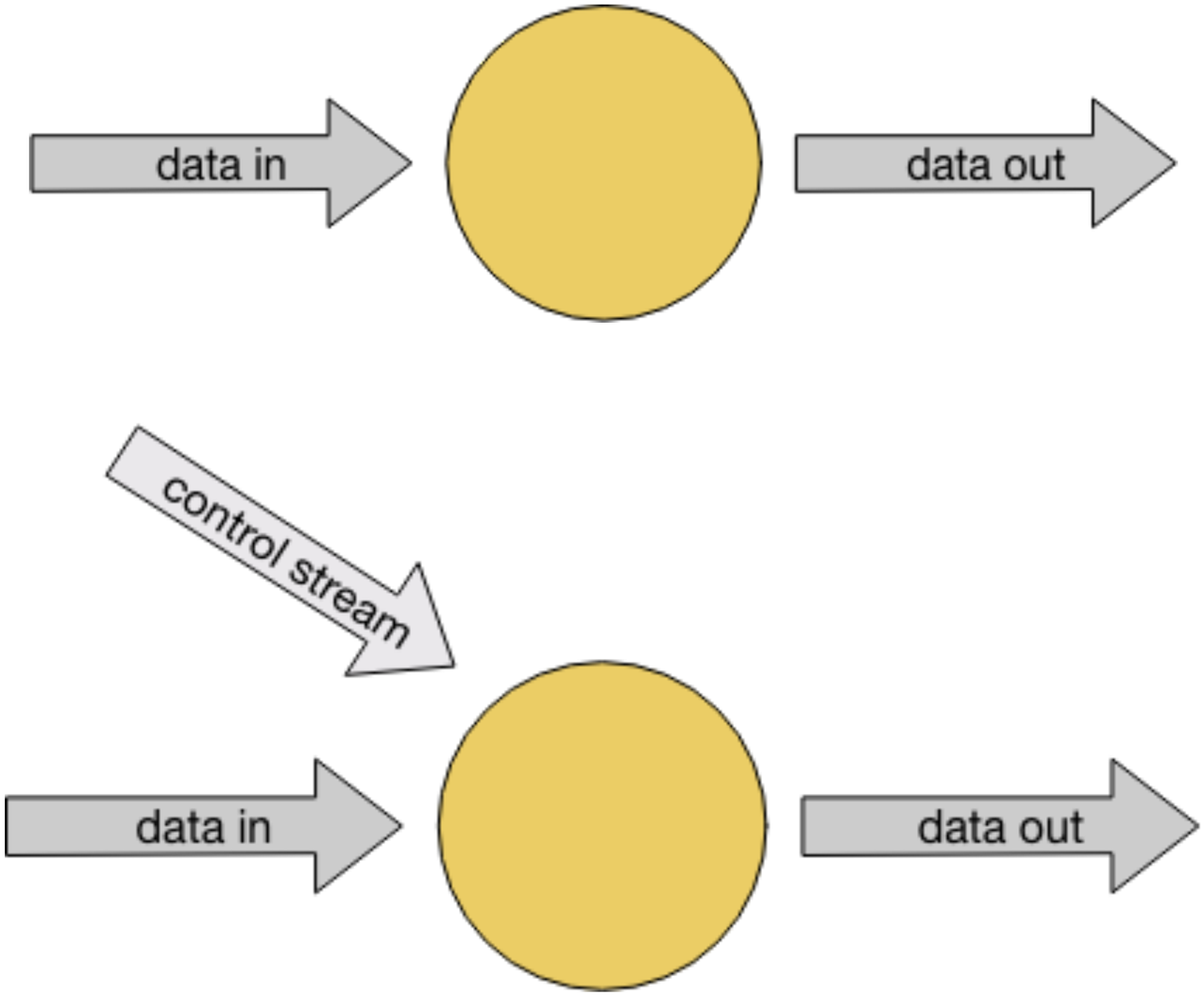
Rich Functions

- open(Configuration c)
- close()
- getRuntimeContext()

```
DataStream<Tuple2<String, Double>> input = …
DataStream<Tuple2<String, Double>> smoothed = input.keyBy(0).map(new Smoother());
```

```
public static class Smoother extends RichMapFunction<Tuple2<String, Double>, Tuple2<String, Double>> {
  private ValueState<MovingAverage> averageState;

  @Override
  public void open (Configuration conf) {
    ValueStateDescriptor<MovingAverage> descriptor =
      new ValueStateDescriptor<>("moving average", MovingAverage.class);
    averageState = getRuntimeContext().getState(descriptor);
  }

  @Override
  public Tuple2<String, Double> map (Tuple2<String, Double> item) throws Exception {
    // access the state for this key
    MovingAverage average = averageState.value();

    // create a new MovingAverage (with window size 2) if none exists for this key
    if (average == null) average = new MovingAverage(2);

    // add this event to the moving average
    average.add(item.f1);
    averageState.update(average);

    // return the smoothed result
    return new Tuple2(item.f0, average.getAverage());
  }
}
```

Connected Streams



```
DataStream<String> control = env.fromElements("DROP", "IGNORE").keyBy(x -> x);
DataStream<String> streamOfWords = env.fromElements("data", "DROP", "artisans", "IGNORE")
        .keyBy(x -> x);

control
    .connect(datastreamOfWords)
    .flatMap(new ControlFunction())
    .print();
```

```
public static class ControlFunction extends RichCoFlatMapFunction<String, String, String> {
  private ValueState<Boolean> blocked;

  @Override
  public void open(Configuration config) {
      blocked = getRuntimeContext().getState(new ValueStateDescriptor<>("blocked", Boolean.class))
  }

  @Override
  public void flatMap1(String control_value, Collector<String> out) throws Exception {
      blocked.update(Boolean.TRUE);
  }

  @Override
  public void flatMap2(String data_value, Collector<String> out) throws Exception {
      if (blocked.value() == null) {
        out.collect(data_value);
      }
  }
}
```

Lab 2 - Stateful Enrichment of Rides and Fares

Time and Analytics

Event Time

- Flink explicitly supports three different notions of time:
  - Event time
  - Ingestion time
  - Processing time (default)

```
final StreamExecutionEnvironment env =
  StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

Watermarks

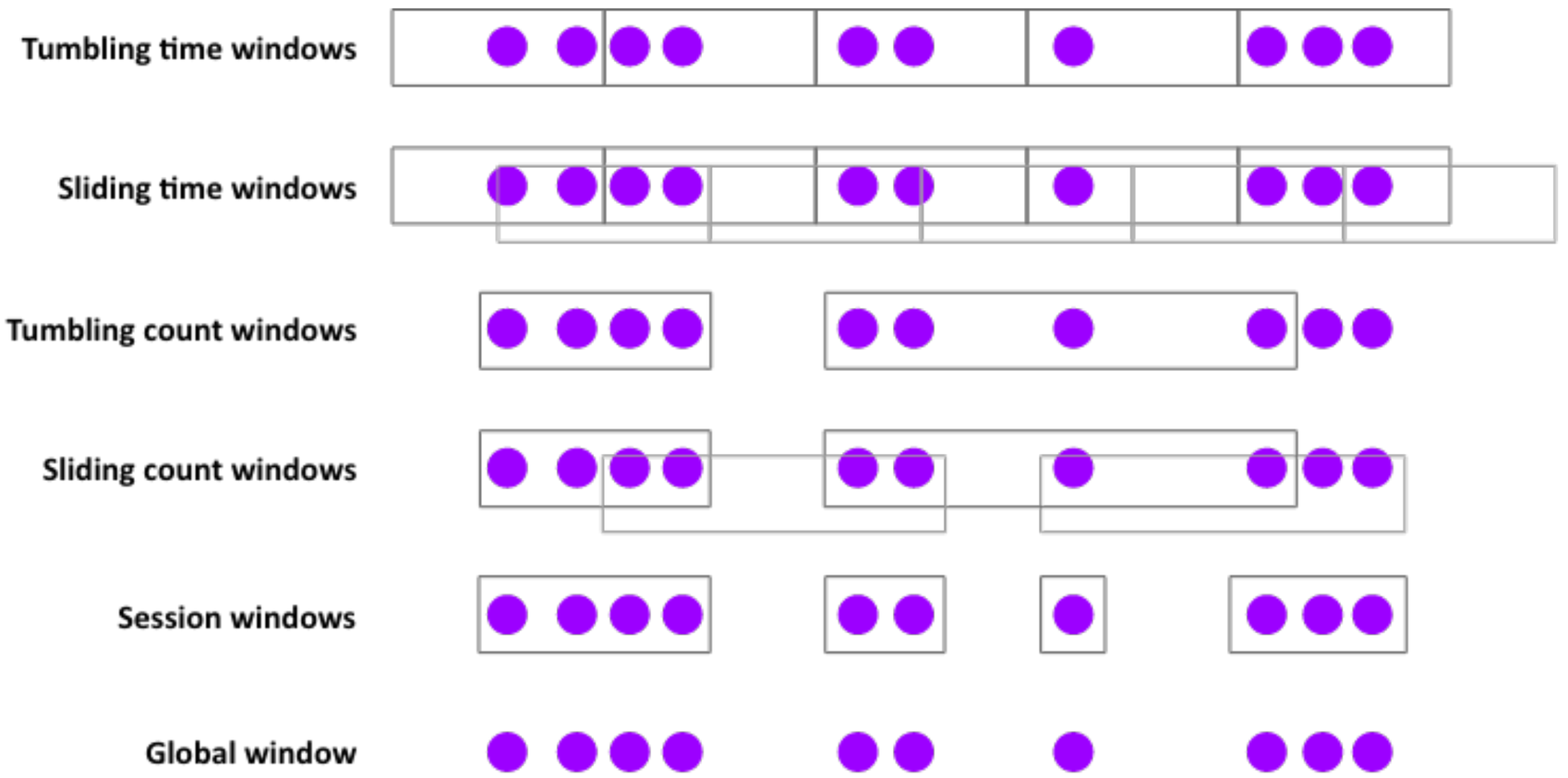
```
··· 23 19 22 24 21 14 17 13 12 15 9 11 7 2 4 →
```

- Data may arrive out of order
- Sorting data is expensive and may not always be required
- Watermark is a good heuristic to bound out of orderness

Watermarks

```
DataStream<MyEvent> stream = ...

DataStream<MyEvent> withTimestampsAndWatermarks =
  stream.assignTimestampsAndWatermarks(new MyExtractor);

public static class MyExtractor
    extends BoundedOutOfOrdernessTimestampExtractor<MyEvent>(Time.seconds(10)) {

  @Override
  public long extractTimestamp(MyEvent event) {
    return element.getCreationTime();
  }
}
```

Windows (Not the OS)



Global Vs Keyed Windows

```
stream.
   .keyBy(<key selector>)
   .window(<window assigner>)
   .reduce|aggregate|process(<window function>)
```

```
stream.
   .windowAll(<window assigner>)
   .reduce|aggregate|process(<window function>)
```

- `TumblingEventTimeWindows.of(Time.minutes(1))`
- `SlidingEventTimeWindows.of(Time.minutes(1), Time.seconds(10))`
- `EventTimeSessionWindows.withGap(Time.minutes(30))`

```
DataStream<SensorReading> input = ...

input
  .keyBy(“key”)
  .window(TumblingEventTimeWindows.of(Time.minutes(1)))
  .process(new MyWastefulMax());

public static class MyWastefulMax extends ProcessWindowFunction<
  SensorReading,                  // input type
  Tuple3<String, Long, Integer>,  // output type
  String,                          // key type
  TimeWindow> {                   // window type

    @Override
    public void process(
      String key,
      Context context,
      Iterable<SensorReading> events,
      Collector<Tuple3<String, Long, Integer>> out) {

        int max = 0;
        for (SensorReading event : events) {
          if (event.value > max) max = event.value;
        }
        out.collect(new Tuple3<>(key, context.window().getEnd(), max));

    }
}
```

Buffers all the events

```
DataStream<SensorReading> input = ...

input
  .keyBy(x -> x.key)
  .window(TumblingEventTimeWindows.of(Time.minutes(1)))
  .reduce(new MyReducingMax(), new MyWindowFunction());

private static class MyReducingMax implements ReduceFunction<SensorReading> {
  public SensorReading reduce(SensorReading r1, SensorReading r2) {
    return r1.value() > r2.value() ? r1 : r2;
  }
}

private static class MyWindowFunction extends ProcessWindowFunction<
  SensorReading, Tuple3<String, Long, SensorReading>, String, TimeWindow> {

  @Override
  public void process(
    String key,
    Context context,
    Iterable<SensorReading> maxReading,
    Collector<Tuple3<String, Long, SensorReading>> out) {

    SensorReading max = maxReading.iterator().next();
    out.collect(new Tuple3<String, Long, SensorReading>(key, context.window().getEnd(), max));
  }
}
```

Precombine

Produce final result

Lateness

- By default, when using event-time windows, late events are dropped.

```
stream.
    .keyBy(...)
    .window(...)
    .allowedLateness(Time.seconds(10))
    .process(...);
```

Lateness #2

- Collect late elements

```
OutputTag<Event> lateTag = new OutputTag<Event>("late"){};

SingleOutputStreamOperator<Event> result = stream.
  .keyBy(...)
  .window(...)
  .sideOutputLateData(late)
  .process(...);

DataStream<Event> lateStream = result.getSideOutput(lateTag);
```

Lab 3 - Hourly Tips

ProcessFunction

```
events.keyBy((ConnectedCarEvent event) -> event.carId)
  .process(new SortFunction())
```

```
public static class SortFunction extends KeyedProcessFunction<String, ConnectedCarEvent, ConnectedCarEvent> {
  /* we'll use a PriorityQueue to buffer not-yet-fully-sorted events */
  private ValueState<PriorityQueue<ConnectedCarEvent>> queueState = null;

  @Override
  public void open(Configuration config) {
      /* set up the state we want to use */
  }

  @Override
  public void processElement(ConnectedCarEvent event, Context context, Collector<ConnectedCarEvent> out) {
    /* add/sort this event into the queue */

    /* set an event-time timer for when the stream is complete up to the event-time of this event */
  }

  @Override
  public void onTimer(long timestamp, OnTimerContext context, Collector<ConnectedCarEvent> out) {
      /* release the items at the head of the queue that are now ready, based on the CurrentWatermark */
  }
}
```

Open()

```
@Override
public void open(Configuration config) {
  ValueStateDescriptor<PriorityQueue<ConnectedCarEvent>> descriptor = new ValueStateDescriptor<>(
    "sorted-events", TypeInformation.of(new TypeHint<PriorityQueue<ConnectedCarEvent>>() {})
  );
  queueState = getRuntimeContext().getState(descriptor);
}
```

processElement()

```
@Override
public void processElement(ConnectedCarEvent event, Context context, Collector<ConnectedCarEvent> out)
throws Exception {
  TimerService timerService = context.timerService();

  if (context.timestamp() > timerService.currentWatermark()) {
    PriorityQueue<ConnectedCarEvent> queue = queueState.value();
    if (queue == null) {
      queue = new PriorityQueue<>(10);
    }
    queue.add(event);
    queueState.update(queue);
    timerService.registerEventTimeTimer(event.timestamp);
  }
}
```

onTimer()

```
@Override
public void onTimer(long timestamp, OnTimerContext context, Collector<ConnectedCarEvent> out)
    throws Exception {
  PriorityQueue<ConnectedCarEvent> queue = queueState.value();
  Long watermark = context.timerService().currentWatermark();
  ConnectedCarEvent head = queue.peek();
  while (head != null && head.timestamp <= watermark) {
    out.collect(head);
    queue.remove(head);
    head = queue.peek();
  }
}
```

SQL

https://github.com/ververica/sql-training

Homework