Apache Flink Hands-On

Stream Processing Deep Dive

Tzu-Li (Gordon) Tai

tzulitai@apache.org

@tzulitai

JCConf Taiwan 2016 #JCConf

What you'll learn today

1. Write basic & advanced Flink streaming programs
2. Learn in-depth, general data streaming concepts
3. Build realistic streaming pipelines with Kafka

Disclaimer:

the material of this workshop is heavily based on

dataArtisans' Flink training exercises

XX Make sure you’re prepared!

- Preparation instructions:
  https://github.com/flink-taiwan/jcconf2016-workshop

- Steps:

1. Clone project to local
2. Fork project to your own Github account
3. Install Intellij IDEA
4. Pull docker container

XX Who am I?

- 戴資力（Gordon）
- Apache Flink Committer
- Co-organizer of Apache Flink Taiwan User Group
- Software Engineer @ VMFive
- Java, Scala
- Enjoy developing distributed computing systems

Flink.tw

Apache Flink Taiwan User Group

- Facebook Group:
  https://www.facebook.com/groups/flink.tw/
- Meetup.com:
  https://www.meetup.com/flink-tw/
- Blog:
  https://blog.flink.tw/

Welcome to join the community :)

A brief introduction of ...

What is Apache Flink?

Apache Flink

an open-source platform for distributed stream and batch data processing

- Apache Top-Level Project since Jan. 2015
- **Streaming Dataflow Engine** at its core
  - Low latency
  - High Throughput
  - Stateful
  - Distributed

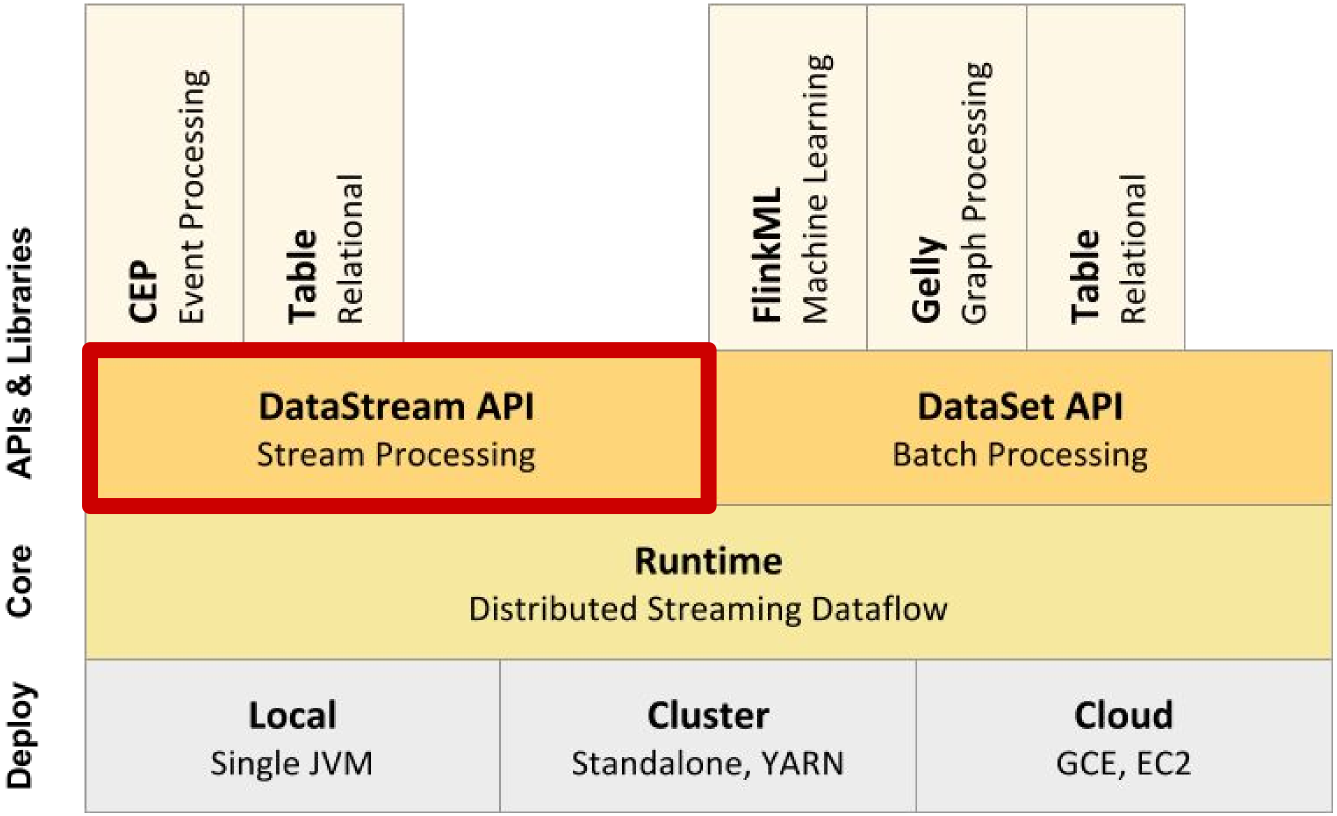
THE APACHE SOFTWARE FOUNDATION

Apache Flink

an open-source platform for distributed stream and batch data processing

- ~230 contributors, 23 Committers / PMCs *(growing)*
- User adoption:
  - **Alibaba** - realtime search ranking optimization
  - **Uber** - ride request fufillment marketplace
  - **Netflix** - Stream Processing as a Service (SPaaS)
  - **Kings Gaming** - realtime data science dashboard
  - **LINE** - realtime log aggregation and system monitoring
  - ...

01 Flink Components Stack



02 Scala Collection-like API

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

DataSet API

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

lines.flatMap(_.split(“ ”)).map(word => Word(word,1))
     .groupBy(“word”).sum(“count”)
     .print()
```

DataStream API

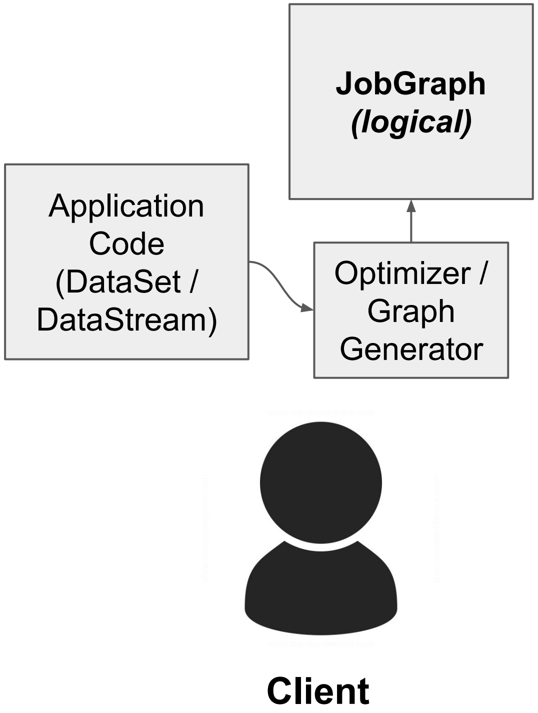
```
val lines: DataStream[String] = env.addSource(new KafkaSource(...))

lines.flatMap(_.split(“ ”)).map(word => Word(word,1))
     .keyBy(“word”).timeWindow(Time.seconds(5)).sum(“count”)
     .print()
```

02 Scala Collection-like API

```
.filter(...).flatmap(...).map(...).groupBy(...).reduce(...)
```

- Becoming the *de facto standard* for new generation API to express data pipelines
- Apache Spark, Apache Flink, Apache Beam ...

03 Flink Programs



JobGraph
(logical)

Application Code (DataSet / DataStream)

Optimizer / Graph Generator

Client

03 Flink Programs

Application code:

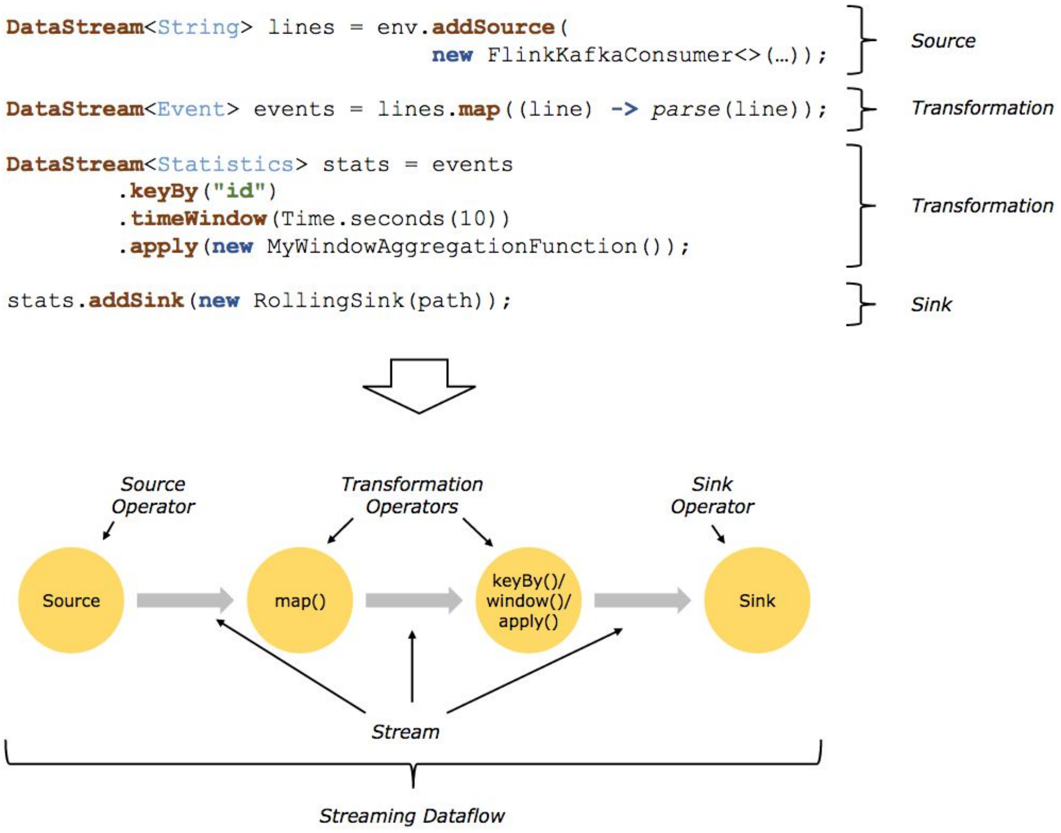
- Define sources
- Define transformations
- Define sinks

optimizer

graph generator

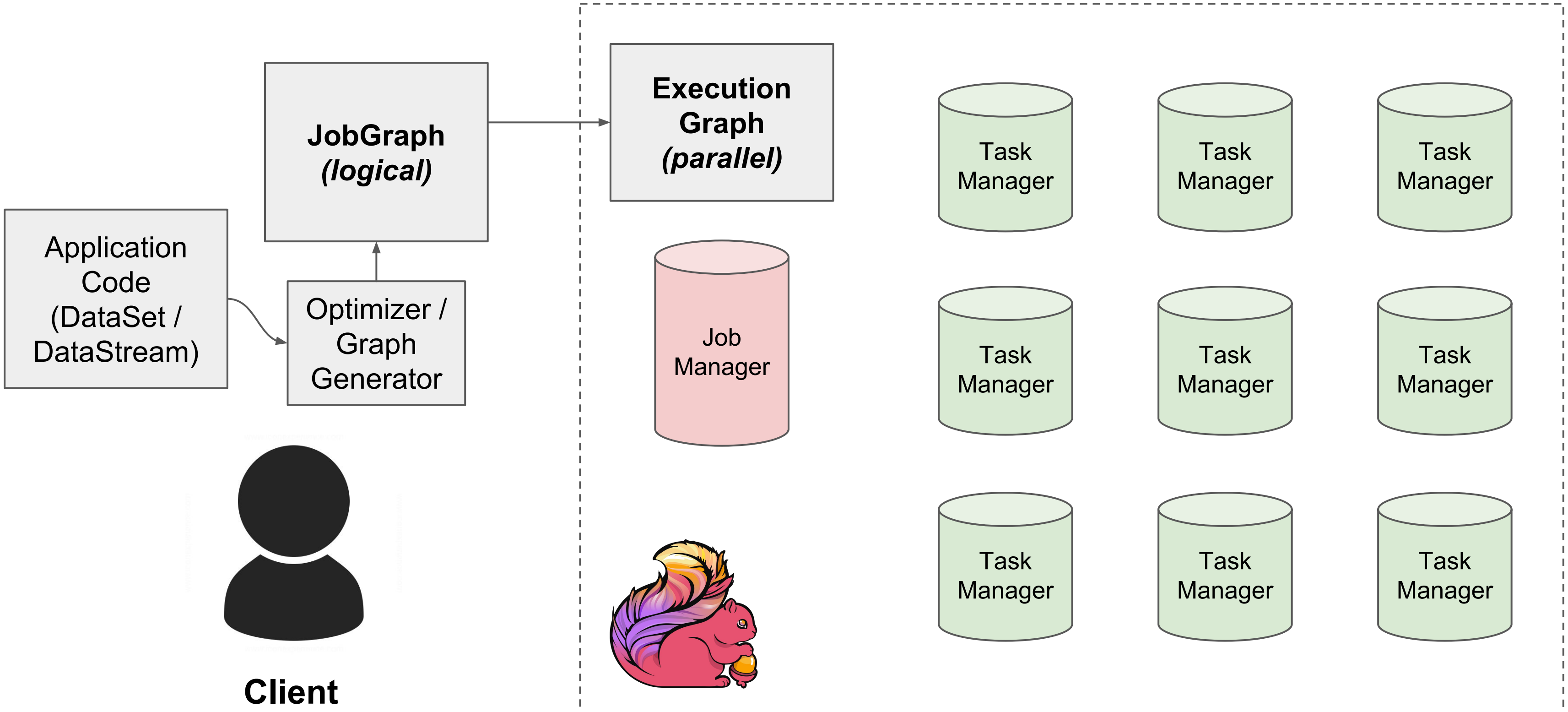
JobGraph

logical view of the dataflow pipeline

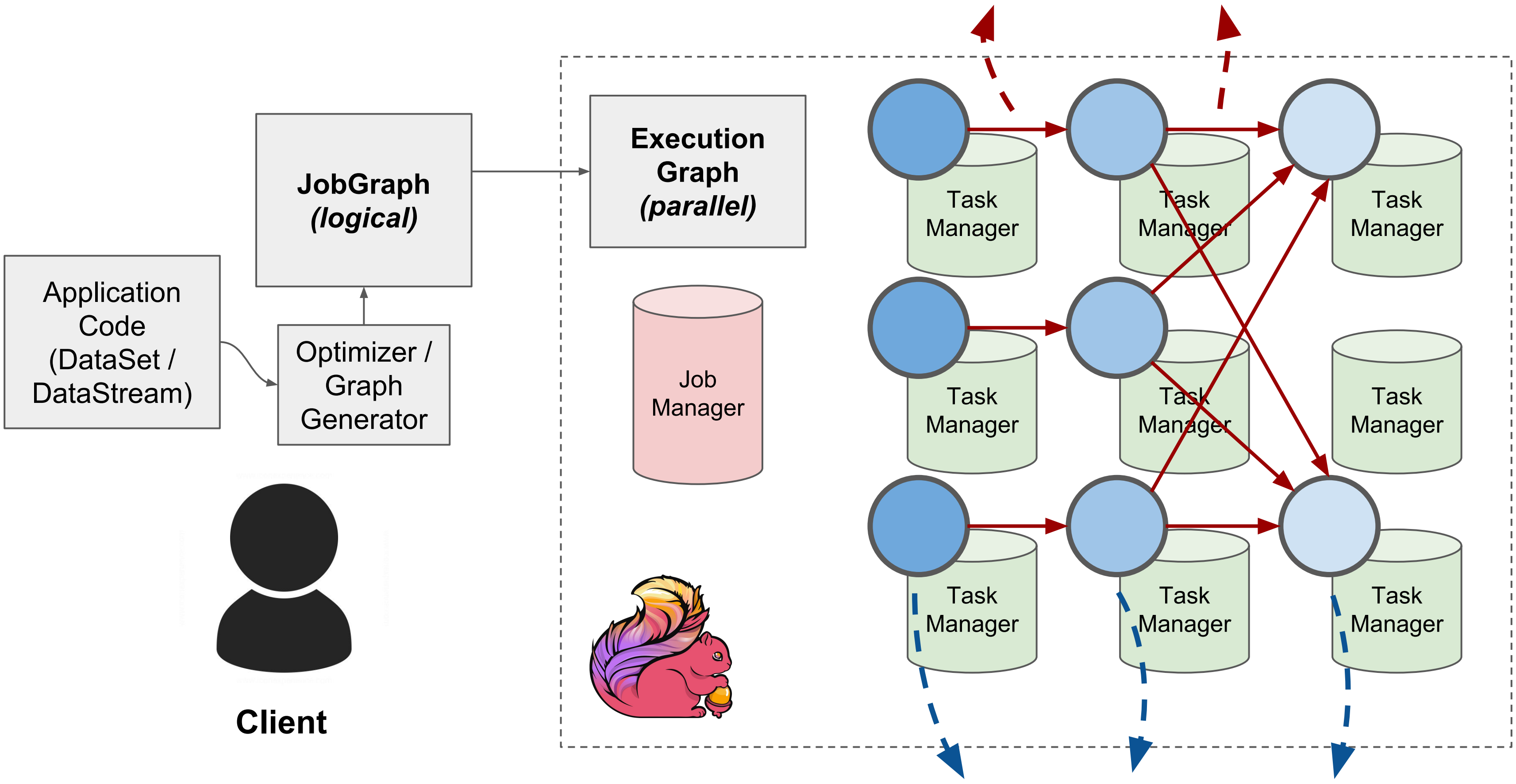


```
DataStream<String> lines = env.addSource(
                                  new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

03 Flink Programs



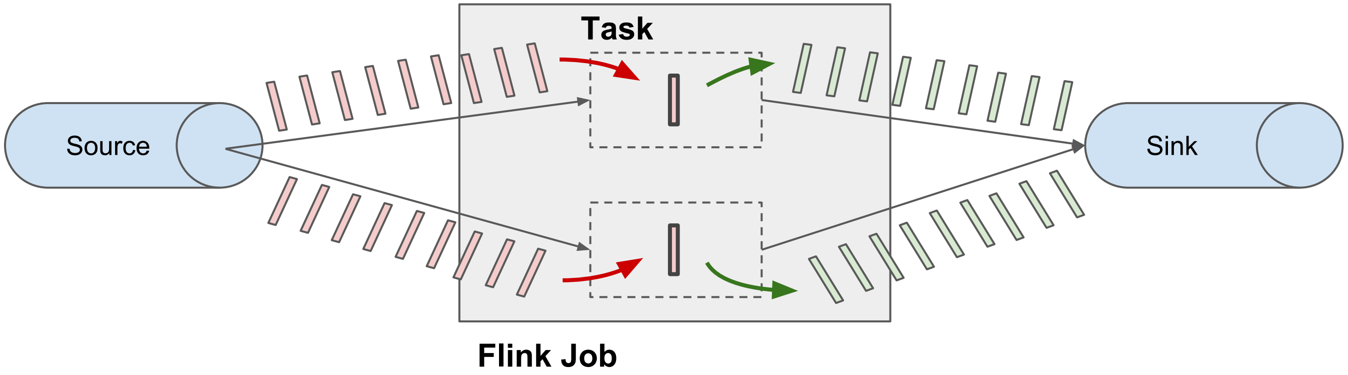
03 Flink Programs


distributed queues as *push-based* data shipping channels
Application Code (DataSet / DataStream)
Optimizer / Graph Generator
JobGraph *(logical)*
Execution Graph *(parallel)*
Job Manager
Client
Task Manager
Task Manager
Task Manager
Task Manager
Task Manager
Task Manager
Task Manager
Task Manager
Task Manager

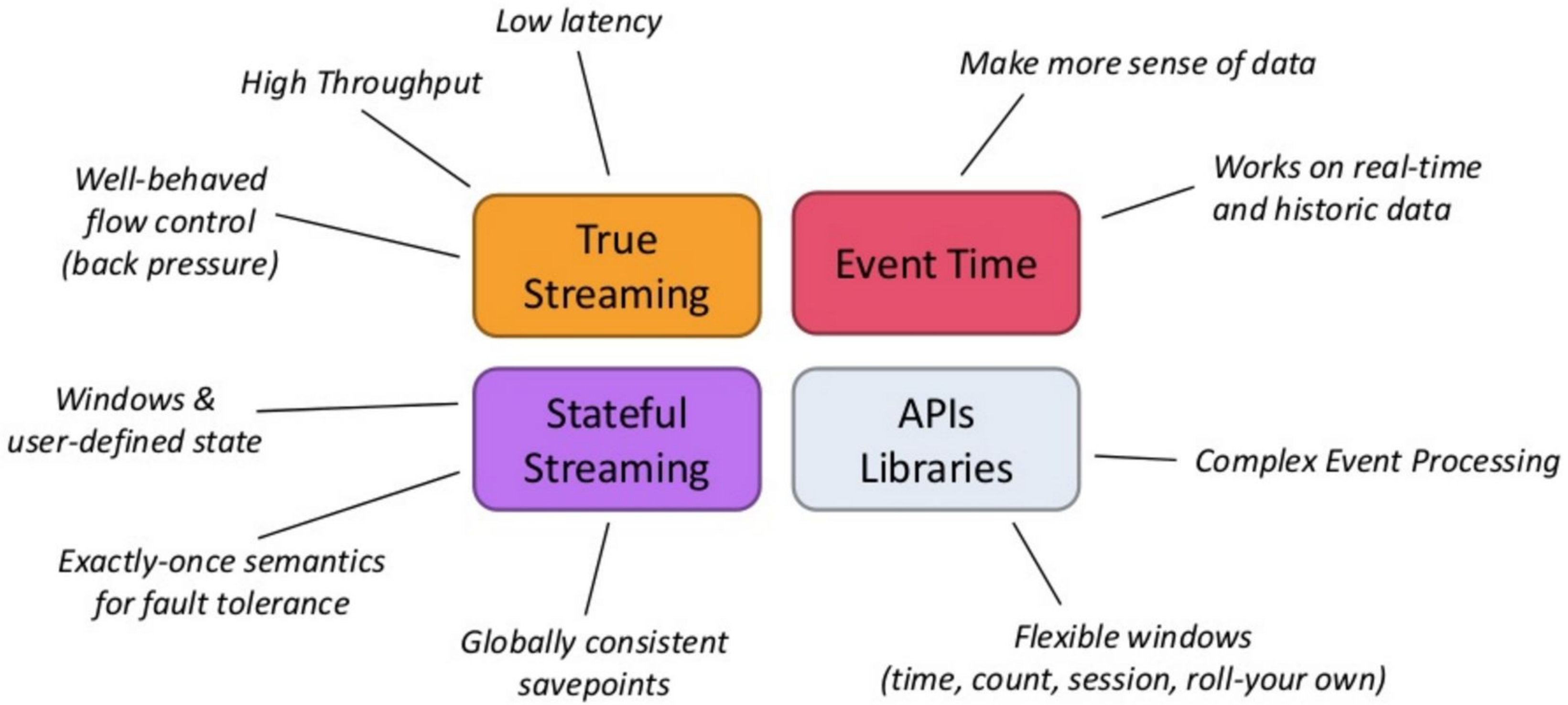
concurrently executed (*even for batch*)

04 Streaming Dataflow Engine

- True one-at-a-time streaming
- Tasks are scheduled and executed concurrently
- Good control of *built-in backpressure*
- Very *flexible windows*
- State is *continuous*



05 Unique Building Blocks


Low latency
High Throughput
Make more sense of data
Well-behaved flow control (back pressure)
True Streaming
Event Time
Works on real-time and historic data
Windows & user-defined state
Stateful Streaming
APIs Libraries
Complex Event Processing
Exactly-once semantics for fault tolerance
Globally consistent savepoints
Flexible windows (time, count, session, roll-your own)

Starting from the basics ...

DataStream API Basics

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

Streaming WordCount: Main Method

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

Stream Execution Environment

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

Data Source

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

Data Types

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

Transformations

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

User-Defined Functions (UDF)

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

Data Sink

```
public static void main(String[] args) throws Exception {
    // get a streaming environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Tuple2<String,Integer>> counts = env
        // read stream of words from socket
        .socketTextStream(“localhost”, 1234)
        // split up the lines into tuple: (word, 1)
        .flatMap(new LineSplitter())
        // use the “word” as key
        .keyBy(0)
        // compute counts every 5 minutes
        .timeWindow(Time.minutes(5))
        // sum up the values
        .sum(1);

    // print result to console
    counts.print();
    // execute program
    env.execute(“Socket Word Count Example”);
}
```

Execute!

LineSplitter: User-Defined FlatMap

```
public static class LineSplitter
    implements FlatMapFunction<String, Tuple2<String,Integer>> {

    @Override
    public void flatMap(String value,
                        Collector<Tuple2<String,Integer>> out)
        throws Exception {
        // normalize and split lines
        String[] words = value.toLowerCase().split( “\\W+”);

        for (String word : words) {
            if (word.length() > 0) {
                out.collect(
                    new Tuple2<String,Integer>(word,  1)
                )
            }
        }
    }
}
```

LineSplitter: User-Defined FlatMap

```
public static class LineSplitter
    implements FlatMapFunction<String, Tuple2<String,Integer>> {

    @Override
    public void flatMap(String value,
                        Collector<Tuple2<String,Integer>> out)
        throws Exception {
        // normalize and split lines
        String[] words = value.toLowerCase().split( “\\W+”);

        for (String word : words) {
            if (word.length() > 0) {
                out.collect(
                    new Tuple2<String,Integer>(word,  1)
                )
            }
        }
    }
}
```

Interface &
Simple Abstract Method

LineSplitter: User-Defined FlatMap

```
public static class LineSplitter
    implements FlatMapFunction<String, Tuple2<String,Integer>> {

    @Override
    public void flatMap(String value,
                        Collector<Tuple2<String,Integer>> out)
        throws Exception {
        // normalize and split lines
        String[] words = value.toLowerCase().split( “\\W+”);

        for (String word : words) {
            if (word.length() > 0) {
                out.collect(
                    new Tuple2<String,Integer>(word,  1)
                )
            }
        }
    }
}
```

FlatMap Collector

06 Other transformations: Map

```
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);

// Map: Takes 1 element, and output 1 element
DataStream<Integer> doubleIntegers =
    integers.map(new MapFunction<Integer, Integer>() {
        @Override
        public Integer map(Integer value) {
            return value * 2;
        }
    });

doubleIntegers.print();
```

```
> 2, 4, 6, 8
```

06 Other transformations: Filter

```
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);

// filter out elements that return false
DataStream<Integer> filtered =
    integers.filter(new FilterFunction<Integer>() {
        @Override
        public boolean filter(Integer value) {
            return value != 3;
        }
    });

filtered.print();
```

```
> 1, 2, 4
```

LineSplitter: User-Defined FlatMap

```
public static class LineSplitter
    implements FlatMapFunction<String, Tuple2<String,Integer>> {

    @Override
    public void flatMap(String value,
                        Collector<Tuple2<String,Integer>> out)
        throws Exception {
        // normalize and split lines
        String[] words = value.toLowerCase().split( “\\W+”);

        for (String word : words) {
            if (word.length() > 0) {
                out.collect(
                    new Tuple2<String,Integer>(word, 1)
                )
            }
        }
    }
}
```

Data Types

07 Flink Type System

- Basic Types
  - Integer, Double, Boolean, String, …
  - Arrays
- Composite Types
  - Tuples
  - Java POJOs
  - Scala case classes

07 Flink Type System: Tuples

- Most easiest and efficient way to encapsulate data
- Scala: default Scala tuples (`Tuple2` to `Tuple22`)
- Java: `Tuple1` to `Tuple25` (Flink’s own implementation)

```
Tuple4<String,String,Integer,Boolean> person =
    new Tuple4<>(“Gordon”, “Tai”, 25, true)

// zero based index
String firstName = person.f0
Integer age = person.f2
```

07 Flink Type System: POJOs

- Any Java class that
  - Has an empty default constructor
  - Has publicly accessible fields (or default getter & setter)

```
public class Person {
    public String firstName;
    public String secondName;
    public int age;
    public boolean isMale;
    public Person() {}
}

DataStream<Person> people = env
    .fromElements(new Person(“Gordon”, “Tai”, 25, true))
```

Hands-On Exercise #1

Taxi Ride Cleansing

08 Keying a Stream

- Keys define how a stream is partitioned and processed by downstream functions:
  - All elements with the same key are processed by the same operator downstream
  - Some operators are key-aware (the input stream must be keyed first, *ex. Windows*)
  - Operator state can be partitioned by key (*more on this later on in the workshop ;)*)

08 Keying a Stream

```
// directly use value index of tuples
DataStream<Tuple2<String,Integer>> wordWithCountStream = ...
wordWithCountStream.keyBy(0)...
wordWithCountStream.keyBy(“f0”)...

// use names of fields in POJO to specify key
DataStream<WordWithCount> wordWithCountStream = ...
wordWithCountStream.keyBy(“word”)...

// can key on multiple fields
DataStream<Tuple3<String,String,Integer>> streamOfTuple3 = ...
streamOfTuple3.keyBy(0,1)...

// or even more flexible, your own key extractor!
DataStream<WordWithCount> wordWithCountStream = ...
wordWithCountStream.keyBy(new KeySelector<>{...})...
```

09 Explicit data distribution

- Besides keys, you can also specify how data is distributed to downstream operators

```
// broadcast to all operators of next transformation
stream.broadcast().map(...);

// round-robin rebalance
stream.rebalance().map(...);

// partition by hash
stream.partitionByHash(...).map(...);

...
```

10 Other transformations: Reduce

```
public static class SumReducer
    implements ReduceFunction<Integer, Integer> {

    @Override
    public Integer reduce(Integer value1, Integer value2)
        throws Exception {
        return value1 + value2;
    }
}
```

Input: [1,2,3,4]

→ **Output:** (((1+2)+3)+4)

11 Working with Multiple Streams

- Connect two streams to correlate them with each other
- Apply functions on connected streams to share state
- Typical use case is to use one stream as *side input* or *control*, and another stream as the data

```
DataStream<Integer> skipLength = ...
DataStream<String> data = ...

DataStream<String> result = skipLength
    .broadcast()
    .connect(data)
    .flatMap(new SkipOrPrintCoFlatMap());
```

11 Working with Multiple Streams

```
public static class SkipOrPrintCoFlatMap
    implements CoFlatMapFunction<Integer,String,String> {

    private Integer lengthToSkip = 0;

    @Override
    public void flatMap1(Integer value, Collector<String> out)
        throws Exception {
        lengthToSkip = 0;
    }

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

Hands-On Exercise #2

Taxi Ride GridCell Toggle

It’s about time for ...

Windows and Time

12 What are Windows for?

- To draw insight from an unbounded stream of data, we need to aggregate *beyond a single record*.

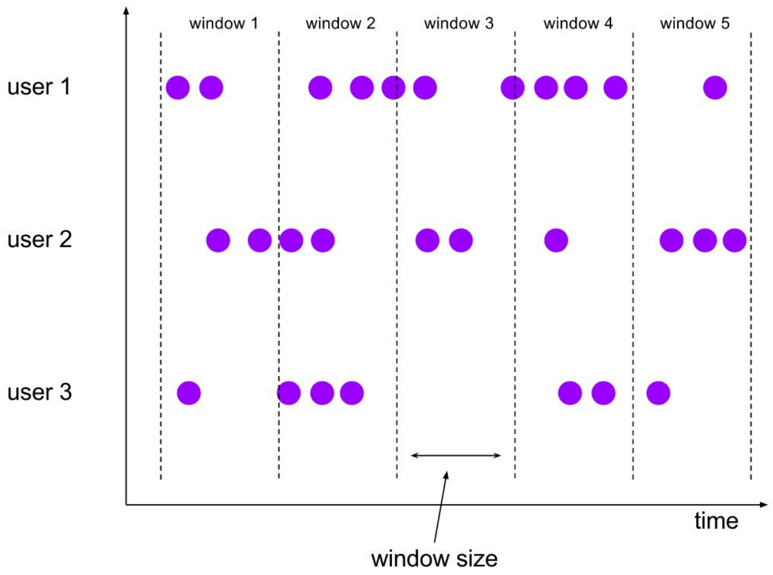
- For example, in the previous streaming WordCount example, we did an aggregation on a *5-minute time window*.

13 Flexible Windows

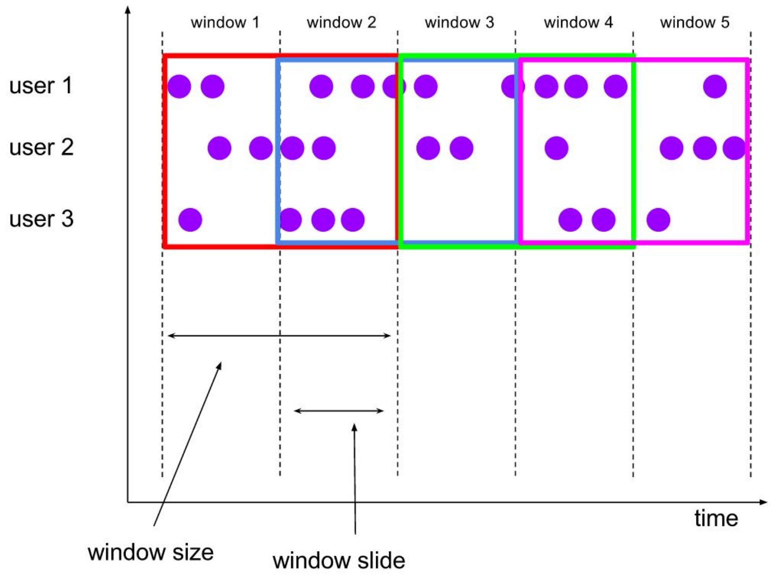
- Due to one-at-a-time processing, Flink has very powerful built-in windowing (certainly among the best in the current streaming framework solutions)
  - Time-driven: *Tumbling window, Sliding window*
  - Data-driven: *Count window, Session window*

14 Time Windows

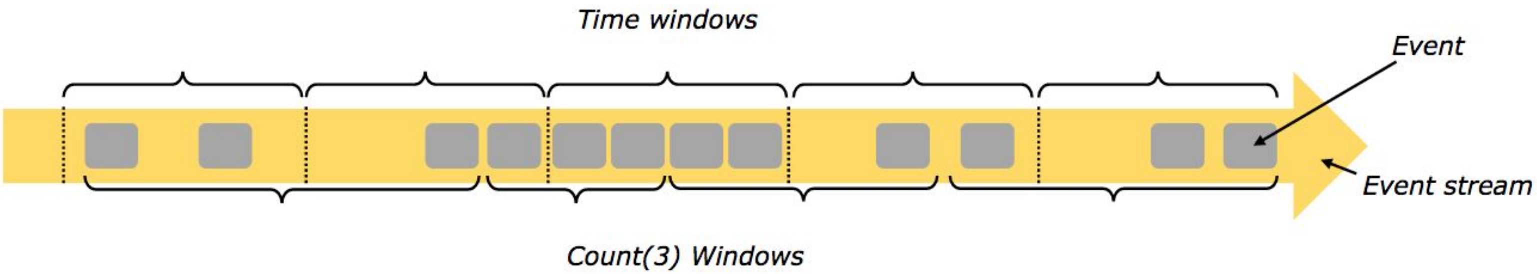
Tumbling Time Window



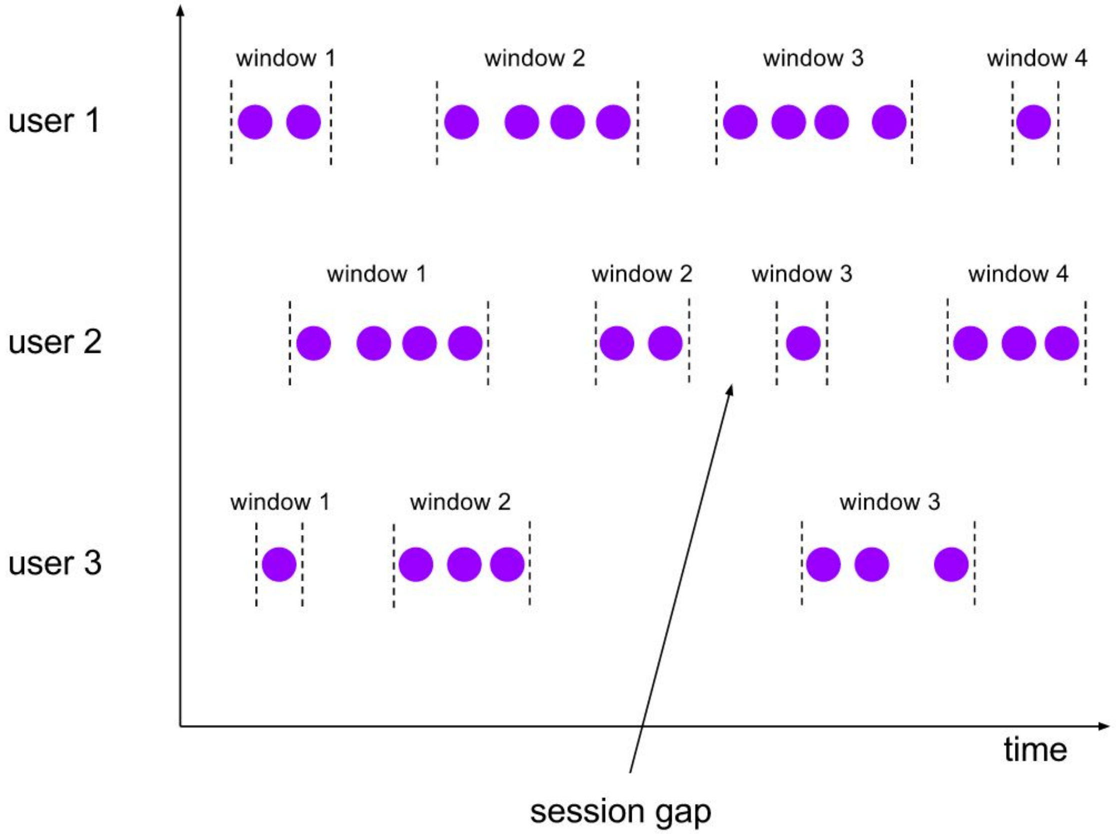
Sliding Time Window



14 Count-Triggered Windows



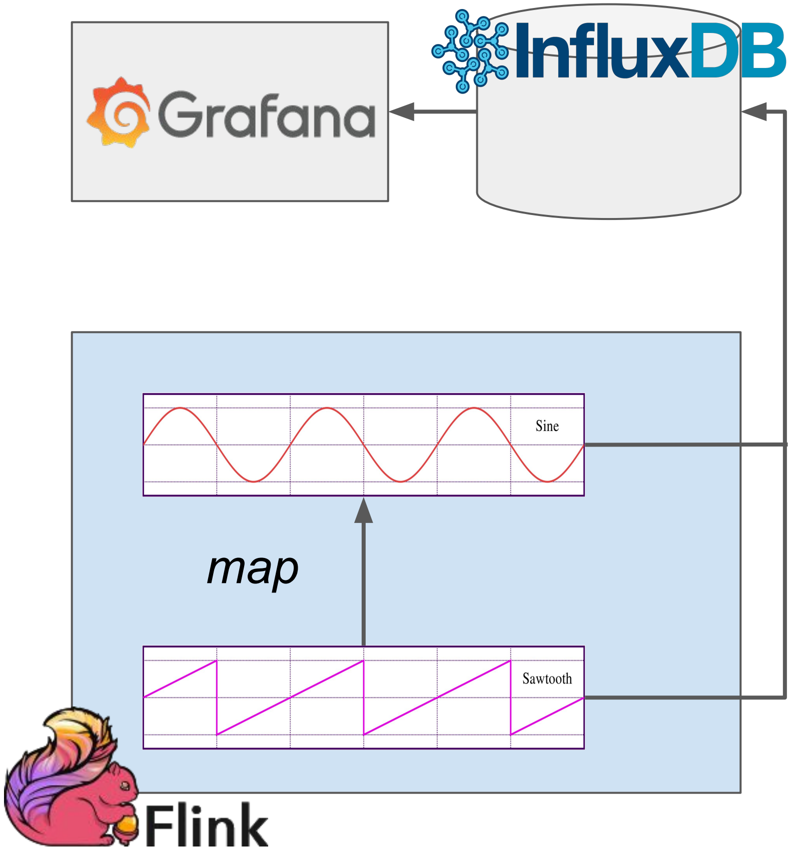
14 Session Windows



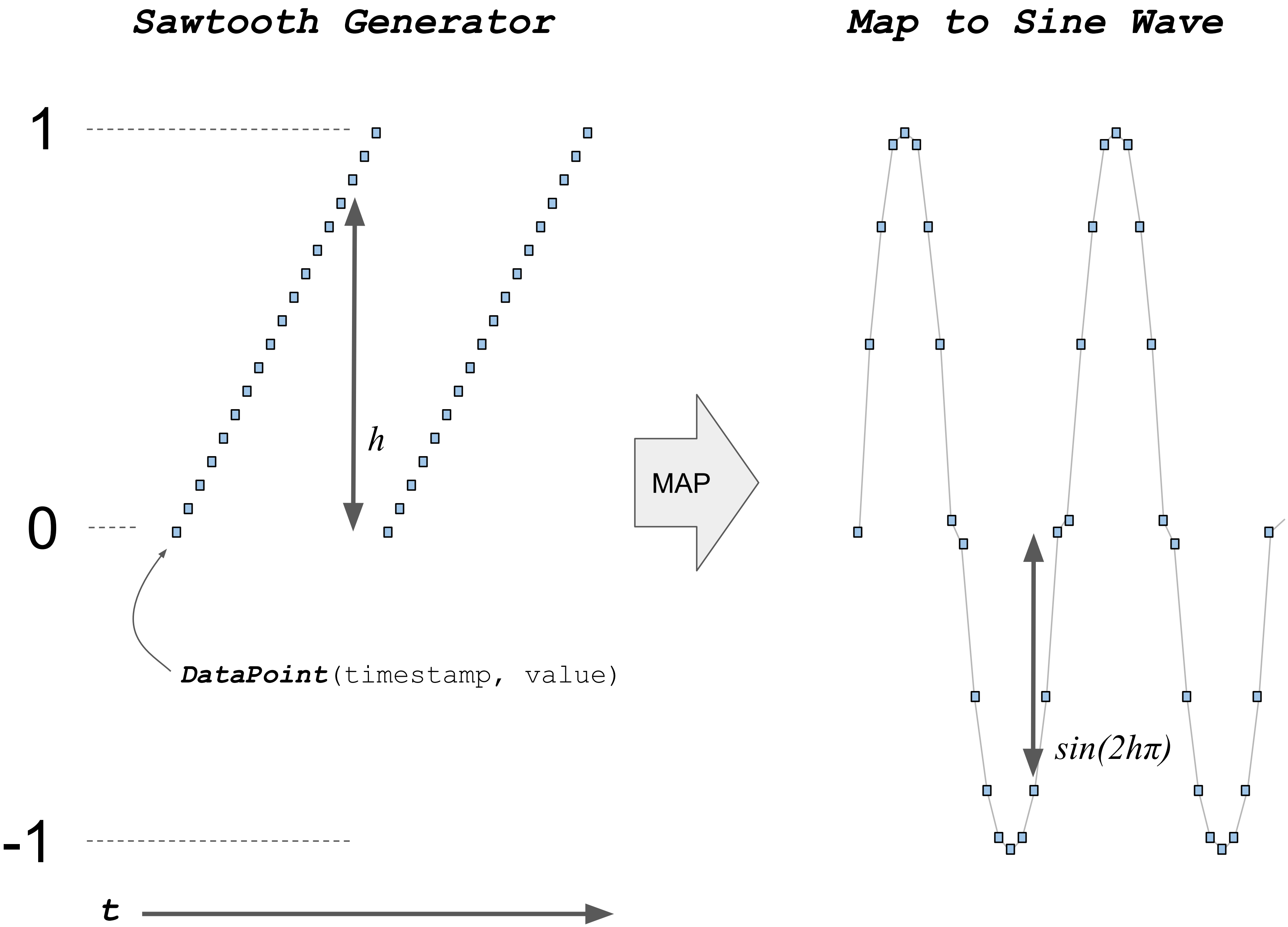
15 Closer look at Time Windows

- Think *Twitter hash-tag count every **5 minutes***
  - We would want the result to reflect the number of Twitter tweets actually tweeted in a 5 minute window
  - ***Not*** the number of tweet events the stream processor receives within 5 minutes

16 Example: Sinewave Sum



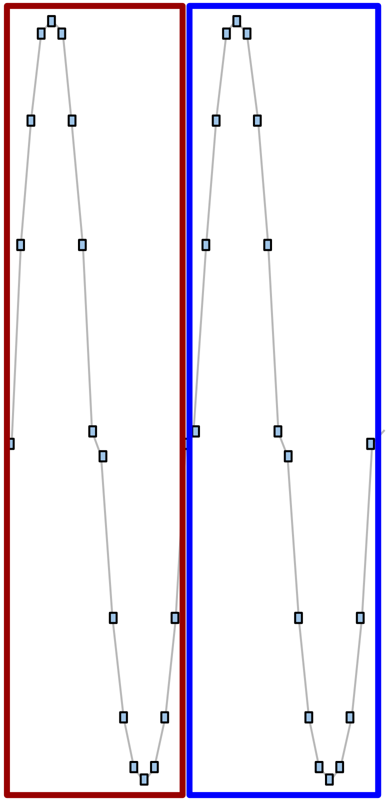
- Generate a sawtooth wave as source
- Map sawtooth to sinewave
- Index both sawtooth & sinewave to InfluxDB (sink)
- Use Grafana to visualize the waves



Sawtooth Generator

Map to Sine Wave

16 Example: Sinewave Sum



tumbling window #1

tumbling window #1

- Perform a tumbling window, with duration as the period of the sine wave

Demo #1

Sinewave Sum

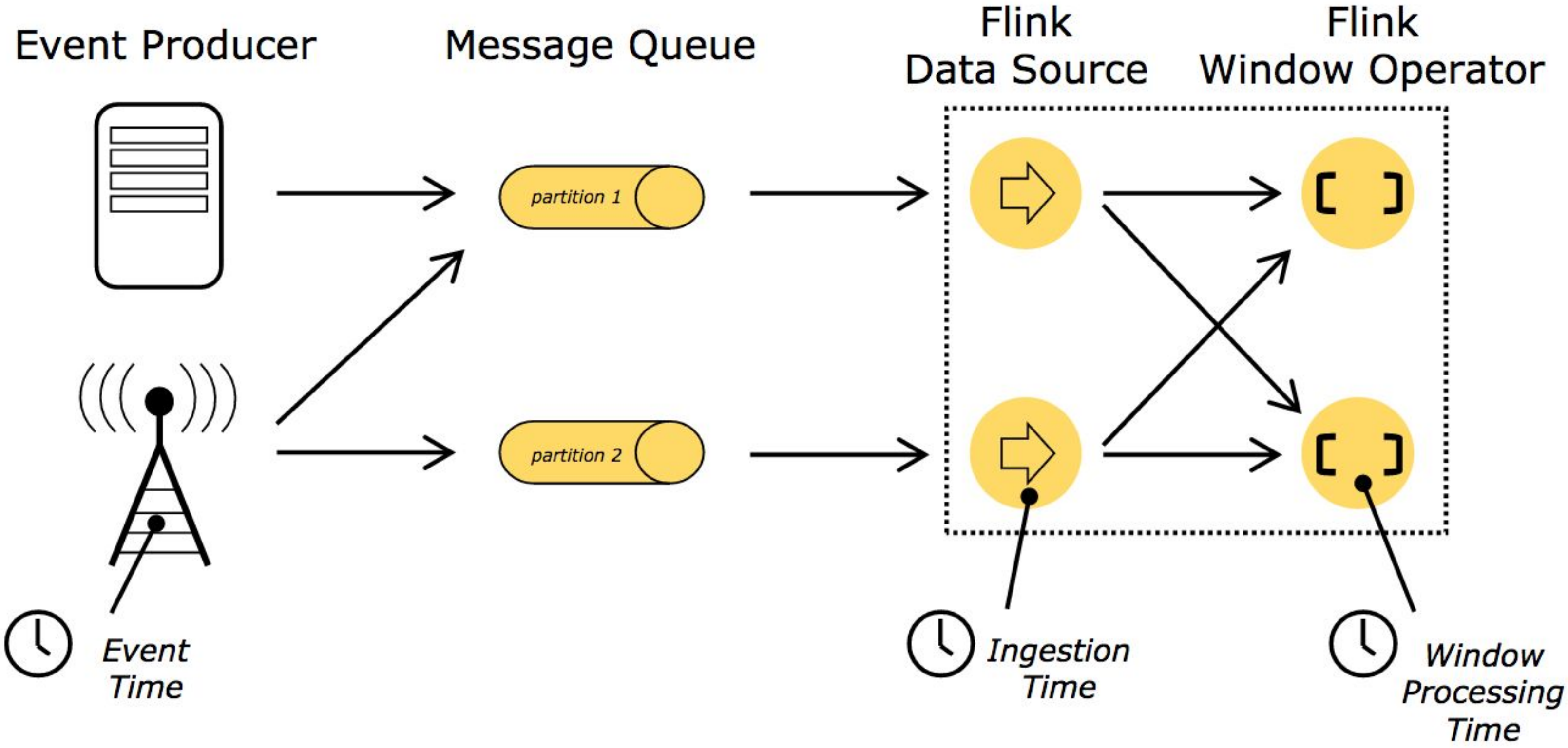
17 Different Kinds of “Time”

- ***Processing Time:***
  - The timestamp at which a system processes an event
  - “Wall Time”

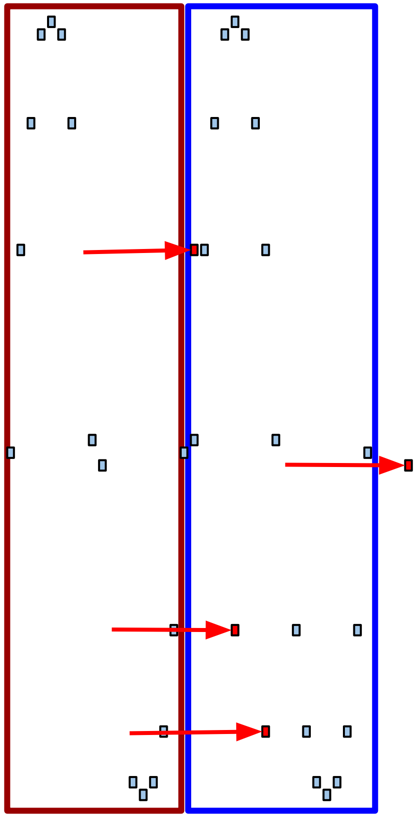
- ***Ingestion Time:***
  - The timestamp at which a system receives an event
  - “Wall Time”

- ***Event Time:***
  - The timestamp at which an event is generated

Different Kinds of “Time”



18 Using Wall Time is Incorrect!



- In reality, data almost never arrives in order
- If the stateful / aggregating / windowing operator works on wall time, the result will definitely be wrong

19 Using Event-Time, in Code

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

env.setTimeCharacteristic(TimeCharacteristic.EventTime);
```

- In reality, you’ll also need to assign the event time to records and emit ***Watermarks*** to help Flink keep track of the event time progress.

- To keep things simple for now, we’ll leave that to after the hands-on practice!

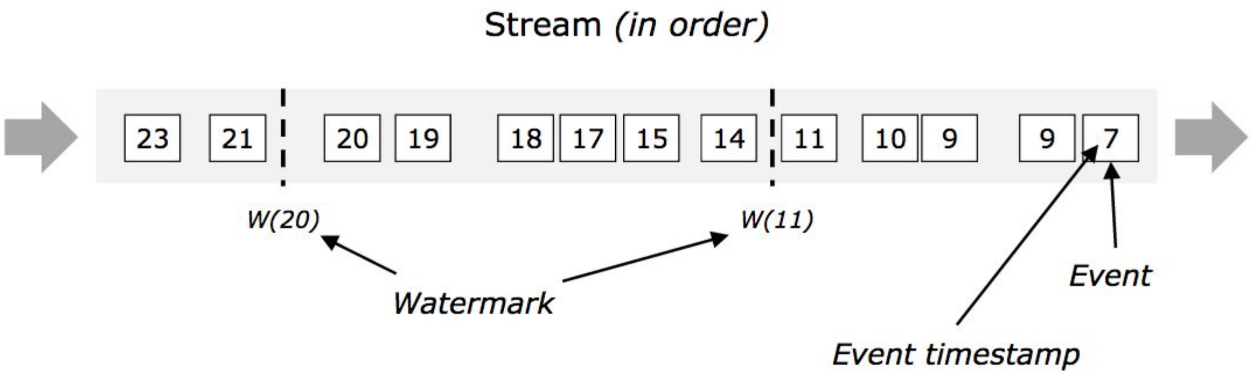
Hands-On Exercise #3

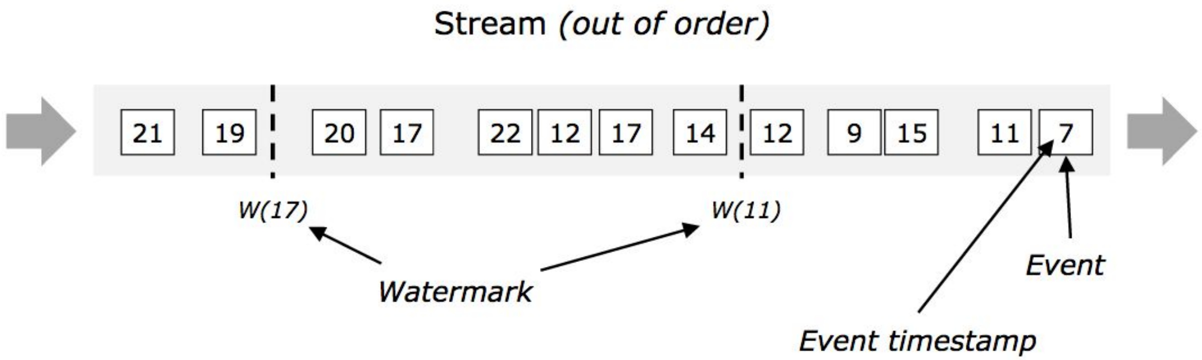
Taxi Ride Popular Places

20 Watermarks & Event-Time

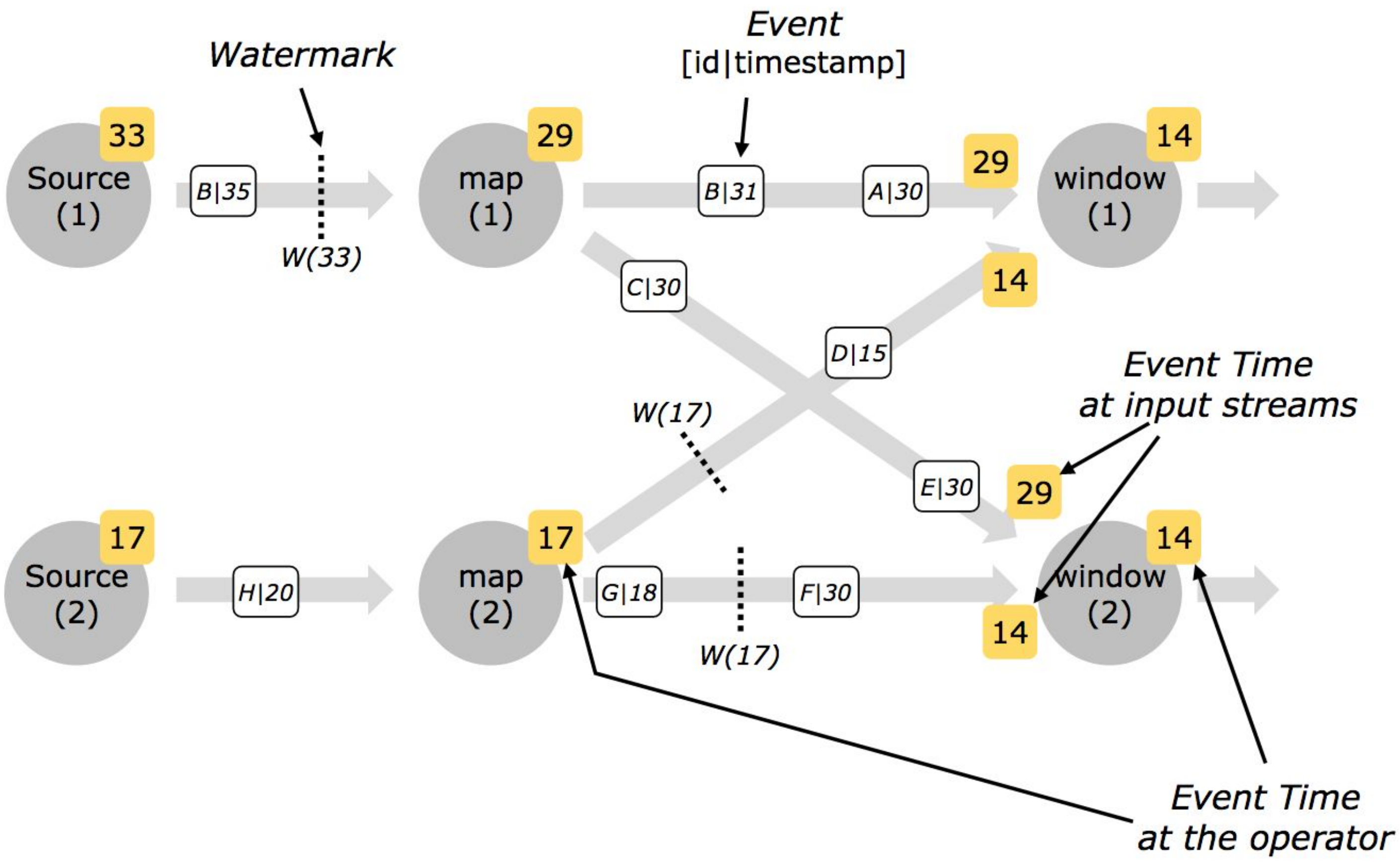
- ***Watermarks*** is a way to let Flink monitor the progress of event time
- Essentially a record that flows within the data stream
- Watermarks carry a timestamp t. When a task receives a t watermark, it knows that there will be no more events with timestamp $t' < t$

20 Watermarks & Event-Time





20 Watermarks & Event-Time



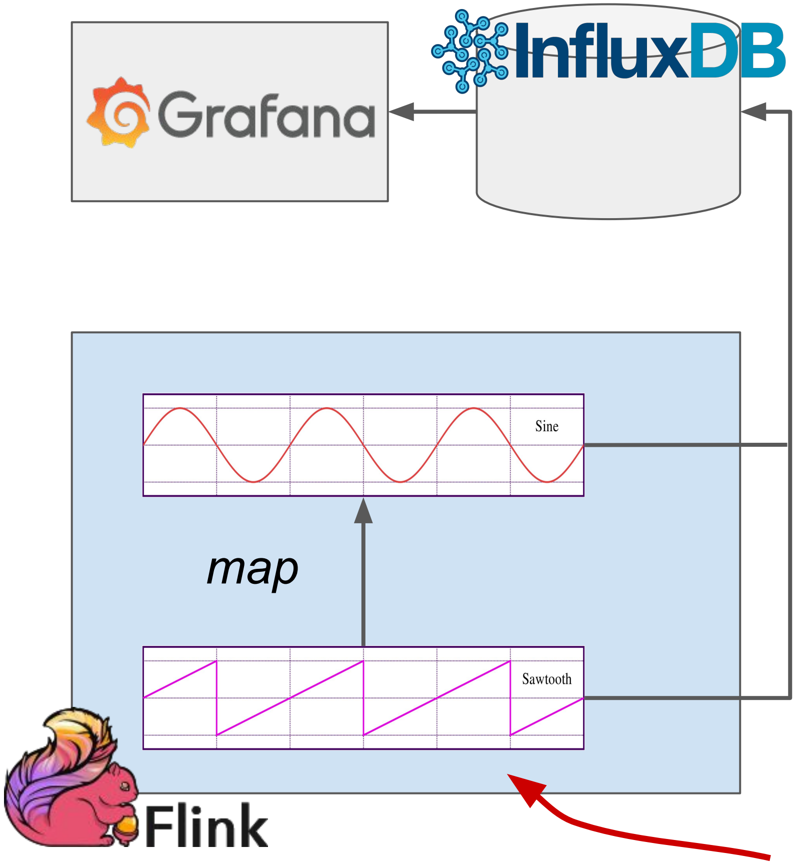
21 Watermarks, in code

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

env.setTimeCharacteristic(TimeCharacteristic.EventTime);

DataStream<Event> events = env.addSource(...);

DataStream<Event> withTimestampsAndWatermarks =
    events.assignTimestampsAndWatermarks(
        new TimestampAndWatermarkAssigner()
    );

withTimestampsAndWatermarks
    .keyBy(...)
    .timeWindow(...)
    .reduce(...)
```

A more realistic streaming pipeline ...

Connecting with Kafka

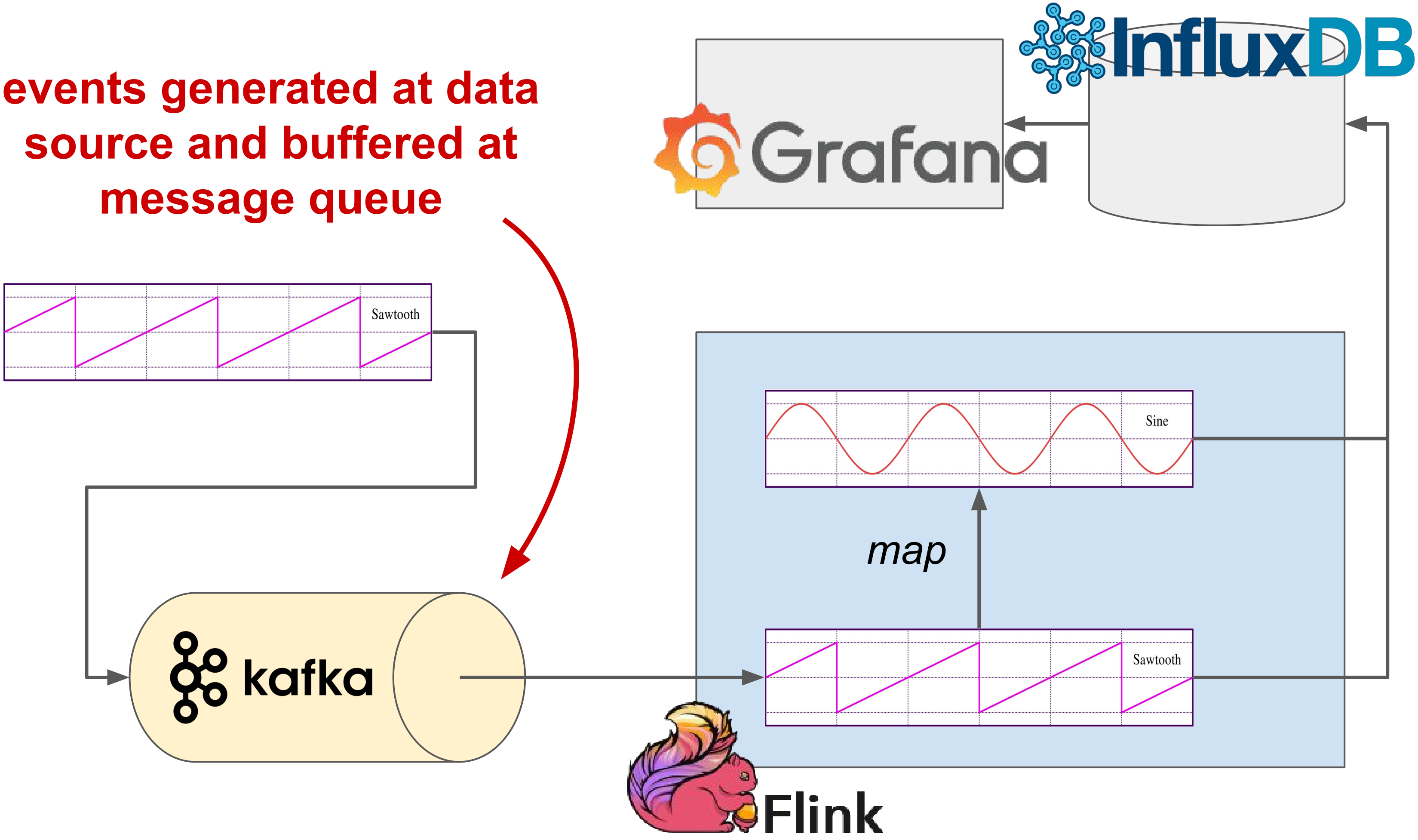
22 The original Sinewave Pipeline



events generated within stream processor (Flink)

23 A More Realistic Pipeline

events generated at data source and buffered at message queue



24 Brief Intro to Apache Kafka

APACHE kafka™

A distributed streaming platform

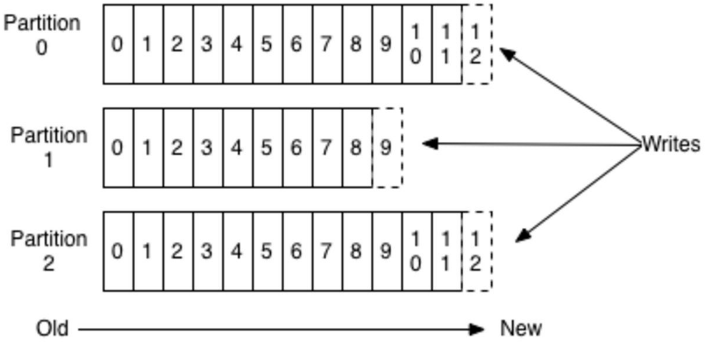
“a publish-subscribe message-queue rethought as a distributed commit log”

- Producers and consumers write and read *topics*.

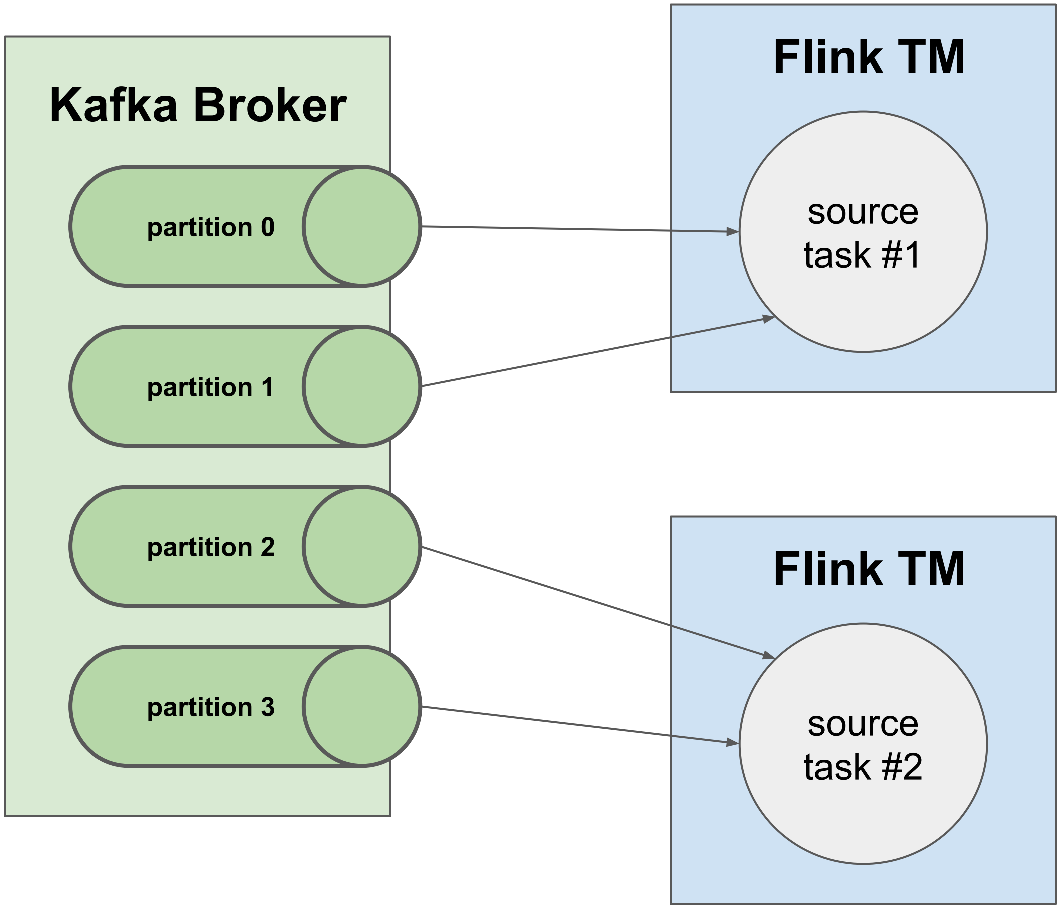
- Each topic consists of many partitions.

- Each record written to a partition has an *offset*.

Anatomy of a Topic



25 How Flink works with Kafka



```
{
  p0 → offset 68,
  p1 → offset 59
}
```

```
{
  p2 → offset 58,
  p3 → offset 61
}
```

Demo #2

Sinewave Pipeline with Kafka

Hands-On Exercise #4

Taxi Ride Pipeline with Kafka

Where Flink shines most!

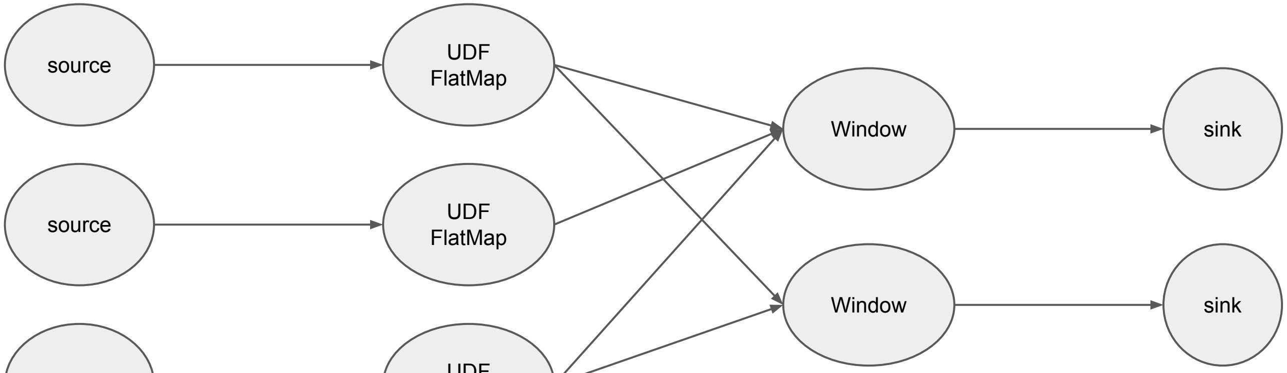
Stateful Streaming

26 Stateful Streaming

- ***Any*** non-trivial streaming application is stateful
- ***Any*** kind of aggregation is stateful (ex. windows)

26 Stateful Streaming

- ***Any*** Flink operator can be stateful
- UDFs can define their own state (*local* or *partitioned*)
- Window operators have built-in state implementation
- Connector sources have "*record offset*" as state



processed record offset

user-defined state

window accumulated state

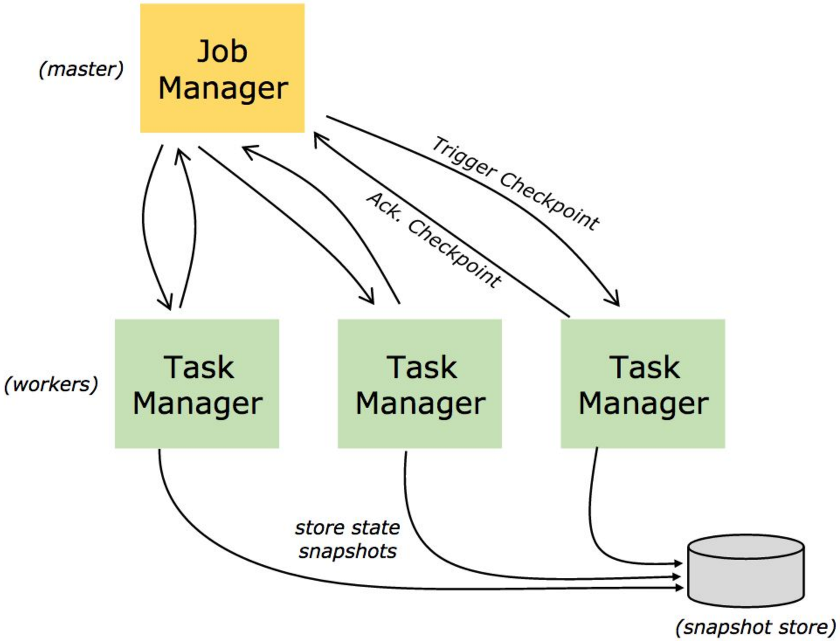
27 Fault Tolerance for States

- What happens if a worker thread for an operator goes down?

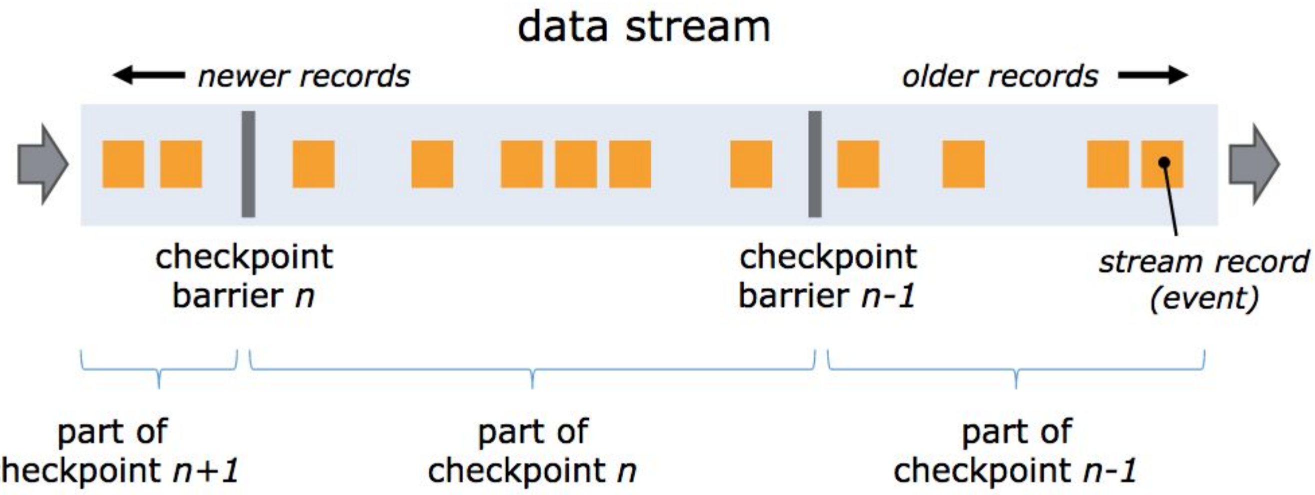
- Different guarantees:
  - **Exactly-once:**
    Each record affects operator state exactly-once
    * *Note: does not mean records are processed only once!*
  - **At-least-once:**
    Each record affects operator state at-least-once

28 Distributed Snapshots

- On each checkpoint trigger, task managers tell all stateful tasks that they manage to snapshot their own state
- When complete, send checkpoint acknowledgement to JobManager
- *Chandy Lamport Distributed Snapshot Algorithm*

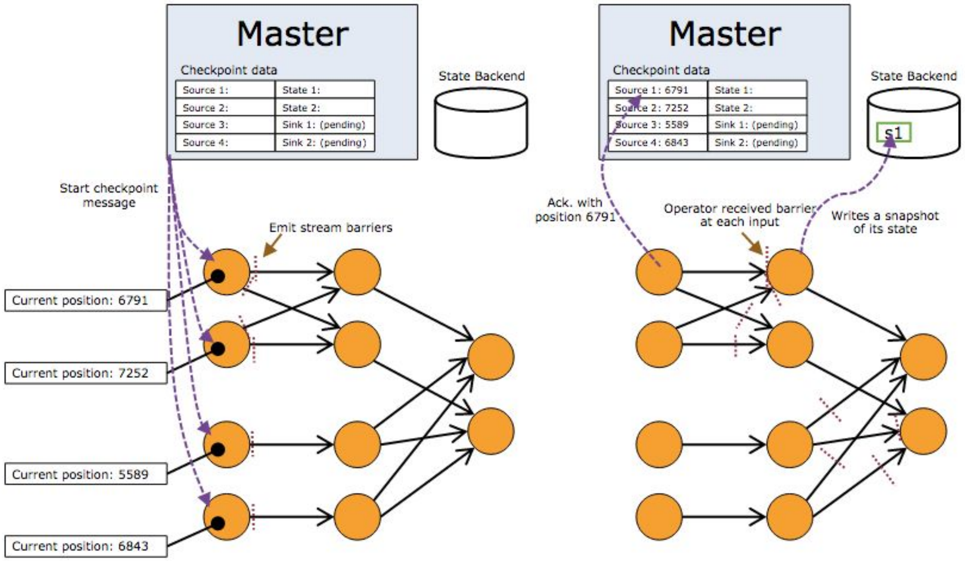


28 Distributed Snapshots



- On a checkpoint trigger by the JobManager, a ***checkpoint*** barrier is injected into the stream

28 Distributed Snapshots



Source 1:	State 1:
Source 2:	State 2:
Source 3:	Sink 1: (pending)
Source 4:	Sink 2: (pending)

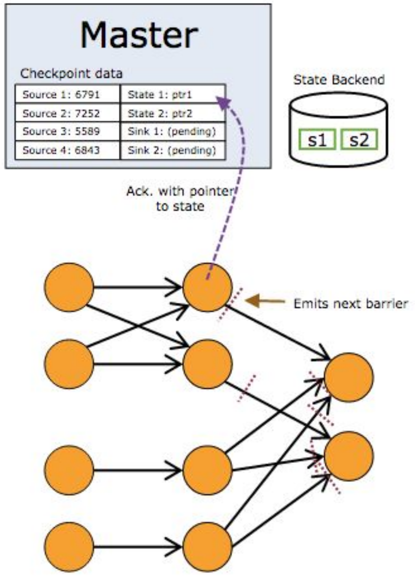
Source 1: 6791	State 1:
Source 2: 7252	State 2:
Source 3: 5589	Sink 1: (pending)
Source 4: 6843	Sink 2: (pending)

Starting Checkpoint

Checkpoint in Progress

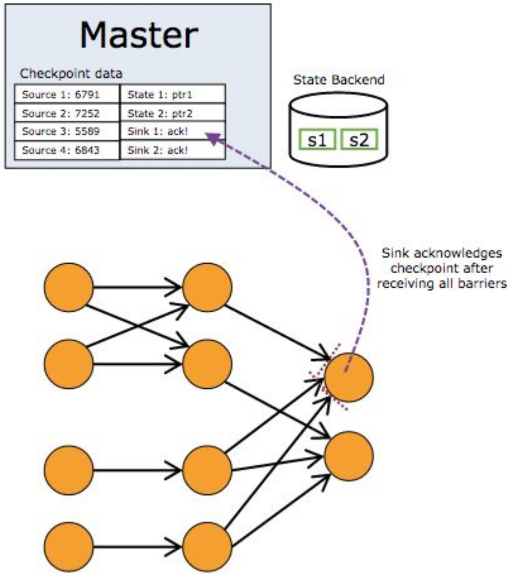
- When a operator receives a checkpoint barrier, its state is checkpointed to a state backend
- A pointer value to the stored state is stored in the distributed snapshot

28 Distributed Snapshots



Source 1: 6791	State 1: ptr1
Source 2: 7252	State 2: ptr2
Source 3: 5589	Sink 1: (pending)
Source 4: 6843	Sink 2: (pending)

Checkpoint
in Progress



Source 1: 6791	State 1: ptr1
Source 2: 7252	State 2: ptr2
Source 3: 5589	Sink 1: ack!
Source 4: 6843	Sink 2: ack!

Checkpoint
Completed

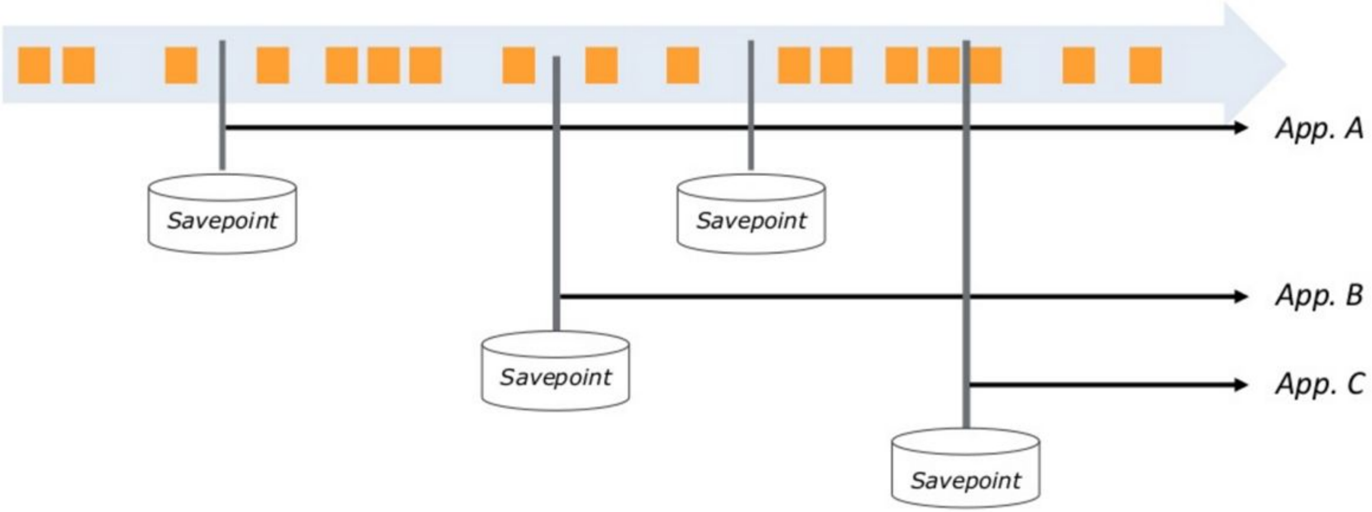
- After all stateful operators acknowledges, the distributed snapshot is completed
- Only fully completed snapshots are used for restore on failure

29 Checkpointing API

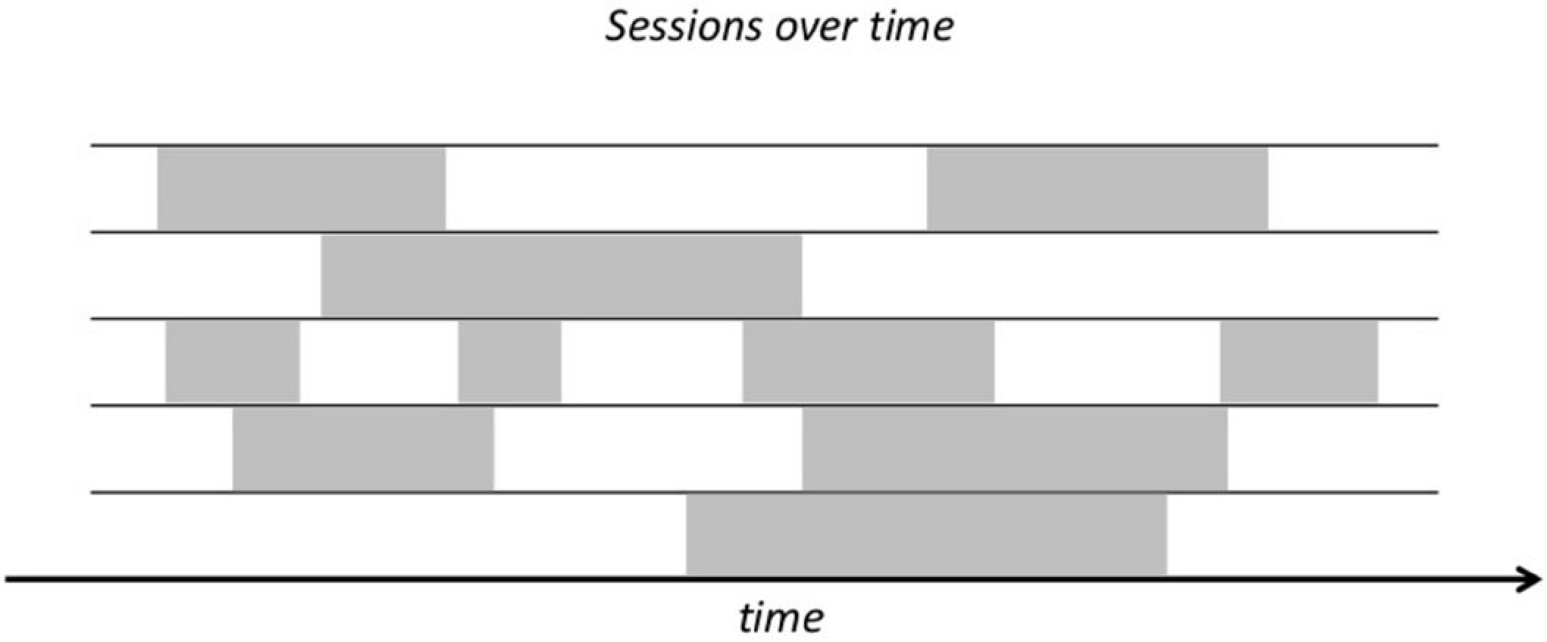
```
final StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();

env.enableCheckpointing(100);
env.setStateBackend(new RocksDBStateBackend(...));
```

30 Flink Streaming Savepoints

- Basically, a checkpointed that is persisted in the state backend
- Allows for stream progress “versioning”

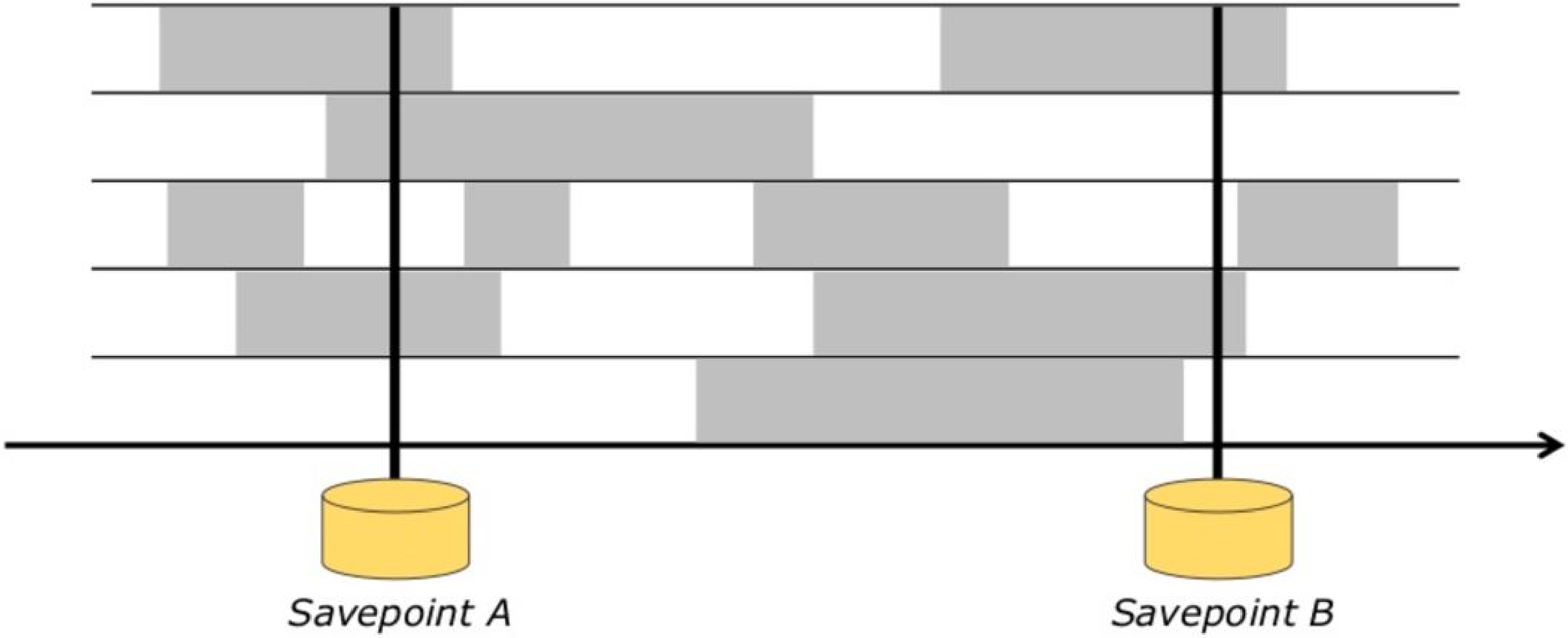


31 Power of Savepoints



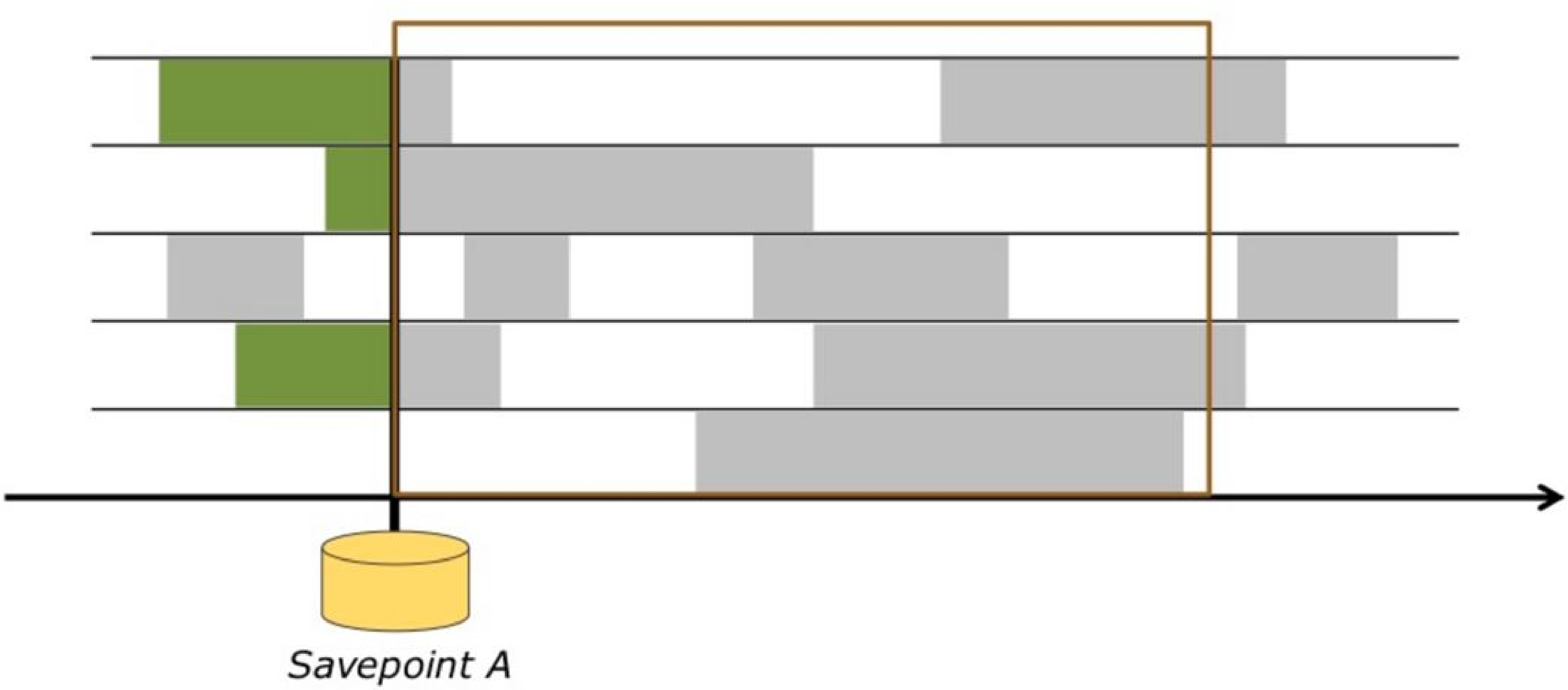
- No stateless point in time

31 Power of Savepoints



- Reprocessing as streaming, starting from savepoint

31 Power of Savepoints



- Reprocessing as streaming, starting from savepoint

Demo #3

Fault Tolerant Sinewave Pipeline

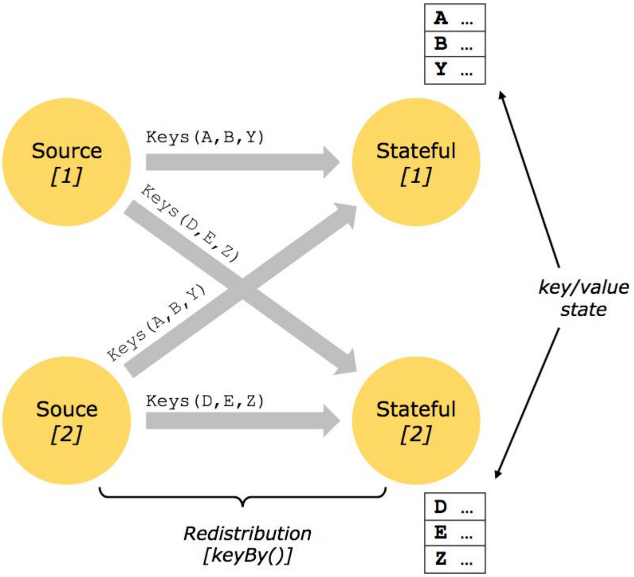
32 Different types of UDF State

- **Local State:** functions can assign any field variable to be checkpointed *(see code for example)*

```
DataStream<String> stream = ...;
DataStream<Long> accumulatedLengths = stream
    .map(new MapToAccumulatedLength());


public static class MapToAccumulatedLength
    implements MapFunction<String, Long>,  Checkpointed<Long> {

    private long accLength = 0;

    @Override
    public Long map(String value) {
        accLength += value.length();
        return accLength;
    }

    @Override
    public void snapshotState(long cpId, long cpTimestamp)
        throws Exception {
        return accLength;
    }

    @Override
    public void restoreState(Long state) {
        accLength = state;
    }
}
```

33 Different types of UDF State

- **Key-Partitioned State:** functions on a keyed stream can access and update state scoped to the current key
  Note: this scales much better and is preferred



→ *State is partitioned with the streams that are read by stateful tasks*

```
DataStream<Tuple2<String,String>> stringsWithKey = ...;
DataStream<Long> accumulatedLengths = stringsWithKey
     .keyBy(0)
     .map(new MapToAccumulatedLength());

public static class MapToAccumulatedLength
     extends RichMapFunction<Tuple2<String, String>, Long> {

     // state object
     private ValueState<Long> accLengthOfKey;

     @Override
     public void open(Configuration conf) {
          // obtain state object
          ValueStateDescriptor<Long> descriptor = new ValueStateDescriptor<>(
               “accLengthOfKey”, Long.class, 0L);
          accLengthOfKey = getRuntimeContext()getState(descriptor);
     }

     @Override
     public Long map(Tuple2<String, String> value) throws Exception {
          long currentLength = accLengthOfKey.value();
          long newLength = currentLength + value.f1.length();
          accLengthOfKey.update(newLength);
          return accLengthOfKey.value;
     }
}
```

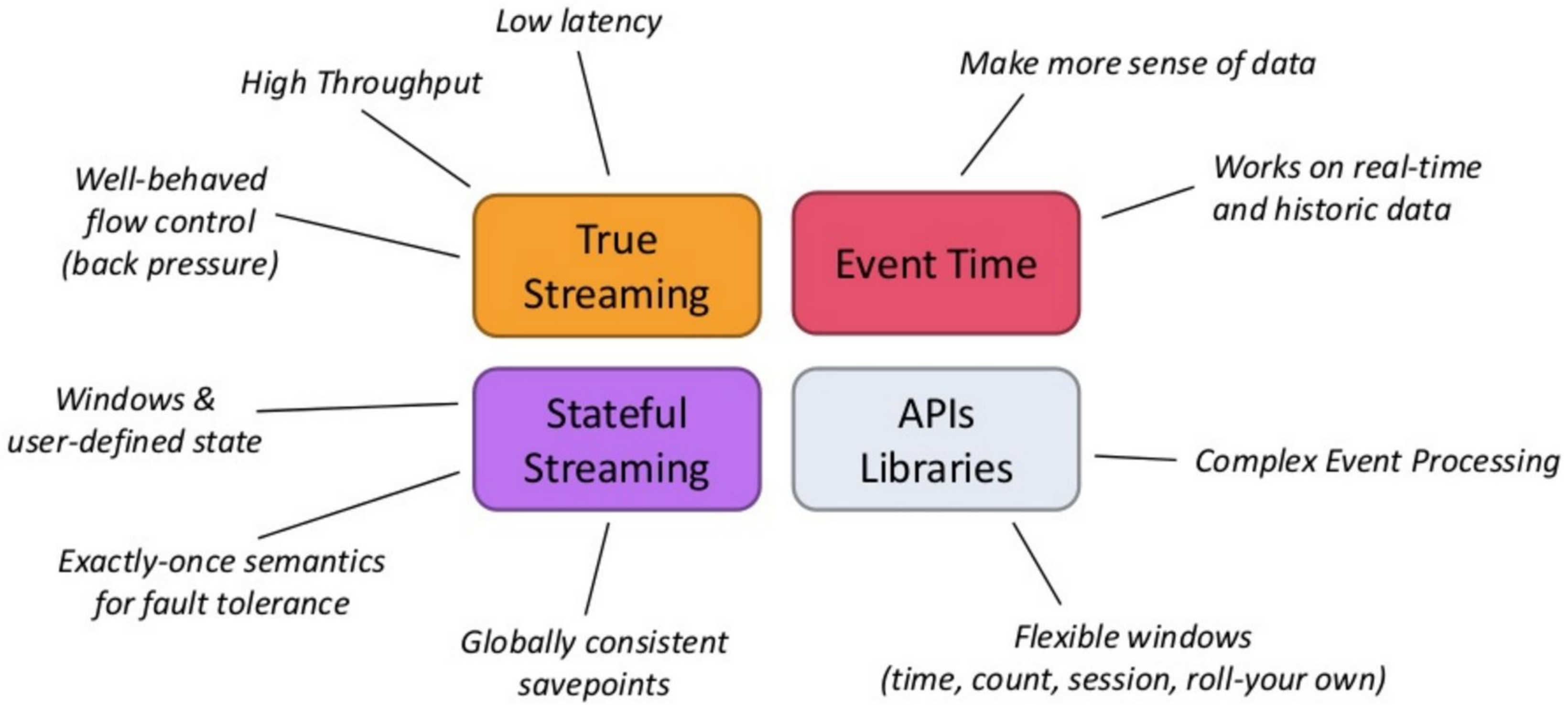
Hands-On Exercise #5

Taxi Ride Duration Prediction

Some final remarks ;)

Conclusion

XX Conclusion


Low latency
High Throughput
Well-behaved flow control (back pressure)
True Streaming
Make more sense of data
Works on real-time and historic data
Event Time
Windows & user-defined state
Stateful Streaming
Exactly-once semantics for fault tolerance
Globally consistent savepoints
APIs Libraries
Complex Event Processing
Flexible windows (time, count, session, roll-your own)

XX Resources

- Apache Flink Documentation:
  https://ci.apache.org/projects/flink/flink-docs-release-1.2/

- dataArtisans Apache Flink Training Material:
  http://dataartisans.github.io/flink-training/

- Apache Flink Taiwan User Group (Facebook):
  https://www.facebook.com/groups/flink.tw/

- Apache Flink Taiwan User Group Meetup.com:
  https://www.meetup.com/flink-tw/