IT5100B

Industry Readiness
Stream Processing

LECTURE 5

Working with Events

FOO Yong Qi
yongqi@nus.edu.sg

CONTENTS

EVENT STREAM PROCESSING

EVENT STREAM PROCESSING

Often we will want to perform some processing on event streams:

Map events

Filter events

Aggregate events

Idea: spin up microservice that consumes events, processes consumed events, and reproduce back into Kafka!

EVENT STREAM PROCESSING

```
KafkaReceiver<Integer, String> IN = getReceiver();
KafkaSender<Integer, String> OUT = getSender();
IN.receive()
    .map(App::myTransform)
    .filter(App::myFilter)
    .map(App::toSenderRecord)
    .doOnNext(x -> OUT.send(x)
            .subscribe());
```

The `Stream` and `Flux` APIs are great for stream processing

EVENT STREAM PROCESSING

STREAM PROCESSING WITH KAFKA

`Stream` and `Flux` lack some features for processing Kafka streams:

- Integration with Kafka cluster?
- Handle consumer group rebalances on stateful operations?
- Stream joins?
- Value updates?

EVENT STREAM PROCESSING

STREAM PROCESSING WITH KAFKA

Idea: expose high-level Domain Specific Language (DSL) for interacting with Kafka Streams

Kafka Streams exposes an incredibly intuitive API for processing Kafka streams, just like how we would process a Stream or Flux

KAFKA STREAMS

KAFKA STREAMS

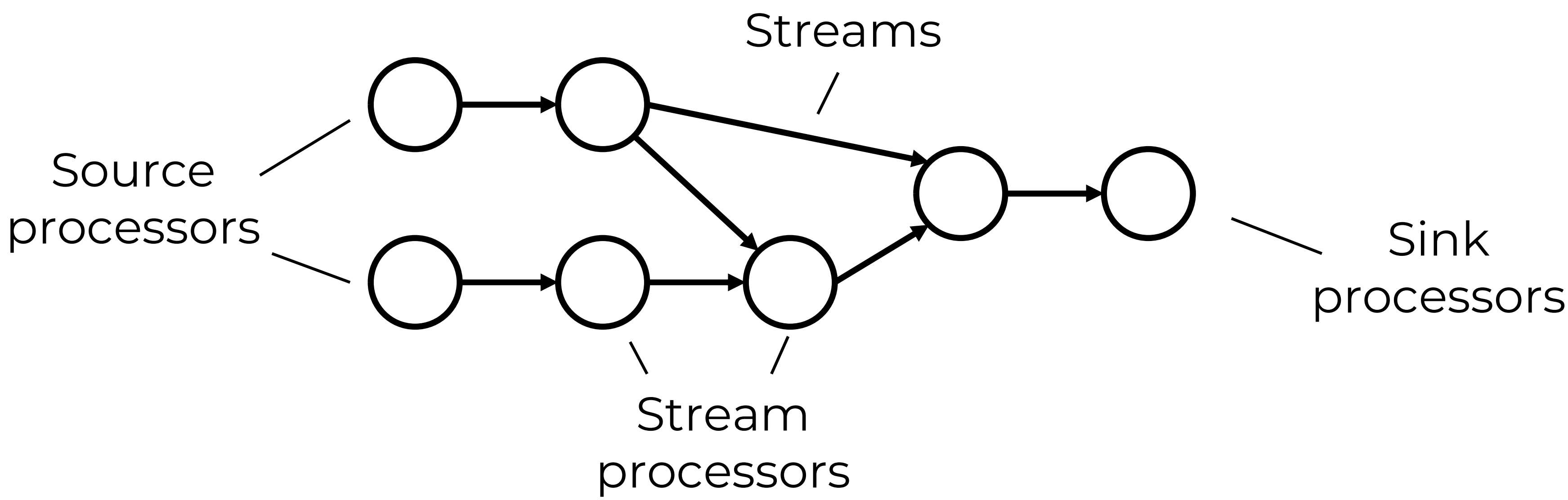
WHAT IS KAFKA STREAMS?

Kafka Cluster

Kafka Streams Application

Java library to write standalone applications (not within the cluster) that interact with Kafka cluster to process data in Kafka

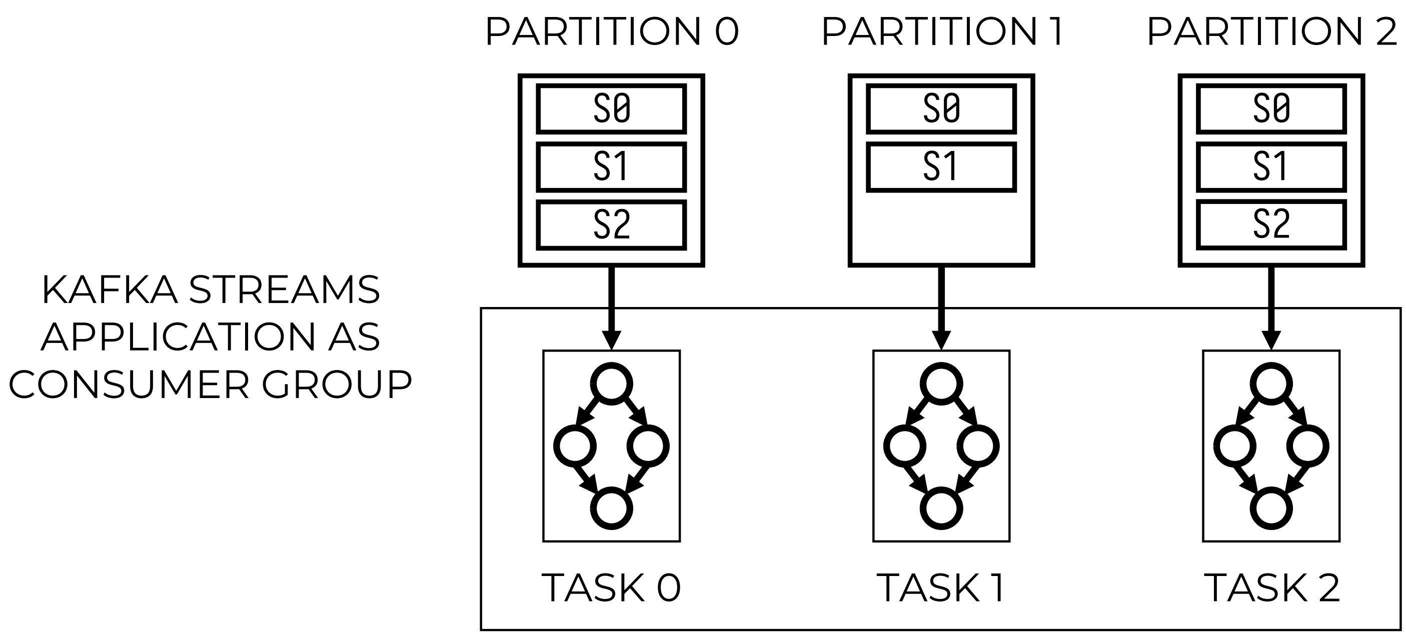
KAFKA STREAMS TOPOLOGY



Kafka Streams application defines **topology** to process streams; each node processes a stream via some operation

KAFKA STREAMS TASKS

PARALLELISM IN KAFKA STREAMS



Kafka Streams are split into tasks; each task is an instance of the topology; each task consumes one partition of the input topic

KAFKA STREAMS TASKS

NOTES ON STREAM TASKS

- **Number of tasks is fixed** upon application instantiation (determined by topics/partitions, not number of app instances)
- Can run multiple application instances
- Each application instance can have multiple threads running in parallel
- Each thread can run multiple tasks
- Excess threads are idle (can be used for fault tolerance)
- If task has multiple input topics, it **consumes same partition number for all topics**

FIRST KAFKA STREAMS PROCESSOR

APPLICATION CONFIGURATION

```
Properties p = new Properties();
p.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-processor-v2");
p.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092,localhost:9093,localhost:9094");
p.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG, StreamsConfig.EXACTLY_ONCE_V2);
p.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 3);
p.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.IntegerSerde.class);
p.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.DoubleSerde.class);
String sourceTopic = "sensor-data-raw";
String sinkTopic = "sensor-data-adjusted";
```

Exactly once processing is simply a configuration parameter

FIRST KAFKA STREAMS PROCESSOR

PROCESSOR TOPOLOGY

```
StreamsBuilder builder = new StreamsBuilder();
KStream<Integer, Double> source = builder.stream(sourceTopic,
        Consumed.with(Serdes.Integer(), Serdes.Double()));
source.peek((k, v) -> System.out.printf("key: %d -> value: %.2f\n", k, v))
        .filter((k, v) -> v >= 0)
        .mapValues(v -> v - 273.15)
        .peek((k, v) -> System.out.printf("key: %d -> new value: %.2f\n", k, v))
        .to(sinkTopic, Produced.with(Serdes.Integer(), Serdes.Double()));
KafkaStreams s = new KafkaStreams(builder.build(), p);
s.start();
```

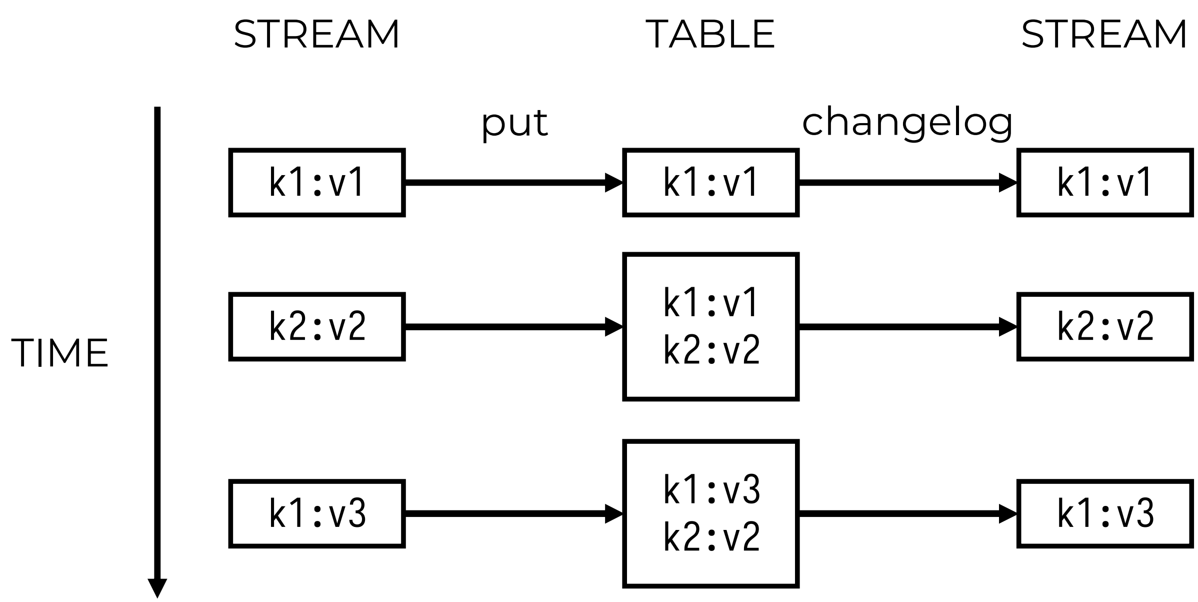
Use **`StreamsBuilder`** to build the processor topology with each node being a `KStream`, then start the processor!

KEY POINT #1

Use the Kafka Streams API to process event streams on Kafka!

STREAM-TABLE DUALITY

STREAM-TABLE DUALITY



Applications require streams and databases (e.g. stream of transactions + database of customer information)

As we have seen, tables are just aggregations of streams!

LOG COMPACTION

UPDATE STREAMS IN KAFKA

Offset	0	1	2	3	4	5	6	7	8
Key	k1	k2	k3	k2	k4	k1	k3	k5	k4
Value	v1	v2	v3	v4	v5	v6	v7	v8	v9

compact

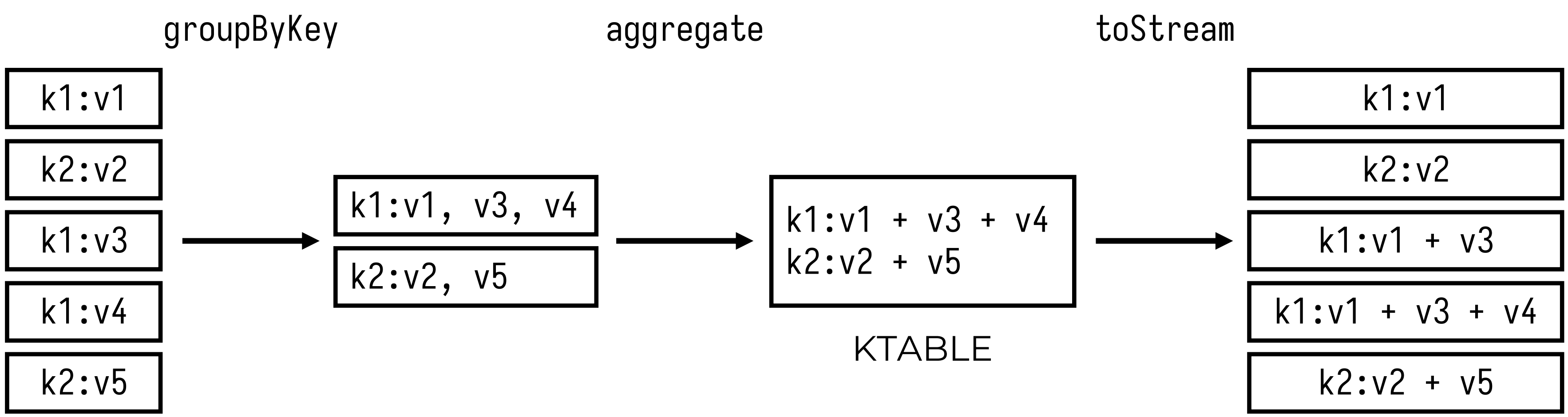
Offset	3	5	6	7	8
Key	k2	k1	k3	k5	k4
Value	v4	v6	v7	v8	v9

Compacted logs only retain latest value of a key of any record; useful as update stream; can be configured per topic

*Add in `--config cleanup.policy=compact` during topic creation to create compacted topic
Consumer offsets are stored in a compacted topic

KTABLES

FIRST-CLASS KAFKA TABLES

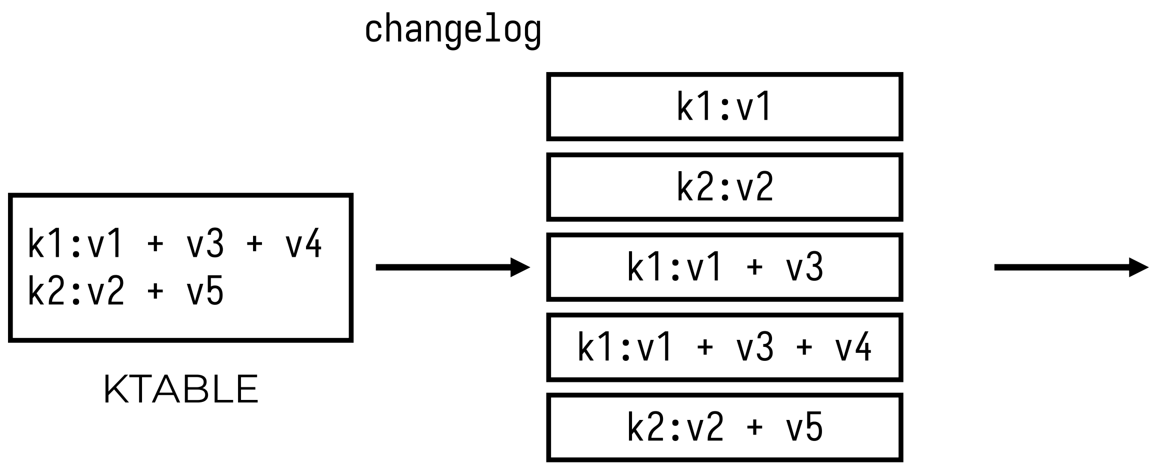


`KTables` are particularly useful for **stateful operations** on events based on keys; `KTables` have a state store backed by RocksDB

`KTables` only read one partition at a time (each task is only assigned one partition), `GlobalKTables` read from entire topic

KTABLES

SCALABLE & FAULT-TOLERANT STATEFUL OPERATIONS



In case of crash/rebalance, `KTable` changelog is backed up as log-compacted topic in Kafka cluster

STATEFUL STREAM PROCESSING

```
Serde<SumCount> sumCountSerde = Serdes.serdeFrom(new SumCountSerializer(),
        new SumCountDeserializer());
KStream<Integer, Double> source = builder.stream(sourceTopic,
        Consumed.with(Serdes.Integer(), Serdes.Double()));
KGroupedStream<Integer, Double> groupedSource = source.groupByKey();
KTable<Integer, SumCount> averager = groupedSource.aggregate(
        SumCount::new,
        (k, v, s) -> s.put(v),
        Materialized.with(Serdes.Integer(), sumCountSerde));
KStream<Integer, Double> averages = averager.mapValues(SumCount::get)
        .toStream();
averages.peek((k, v) -> System.out.printf("%d: %.2f\n", k, v))
        .to(sinkTopic, Produced.with(Serdes.Integer(), Serdes.Double()));
KafkaStreams s = new KafkaStreams(builder.build(), p);
s.start();
```

Now we can perform stateful operations easily without caring about consumer group rebalancing and fault tolerance!

KEY POINT #2

Use KTables for fault-tolerant and scalable stateful stream processing!

TIME

TIMESTAMPS

Optional headers
Key
Value
Timestamp

Timestamps in Kafka events drive Kafka Streams; two kinds of time:

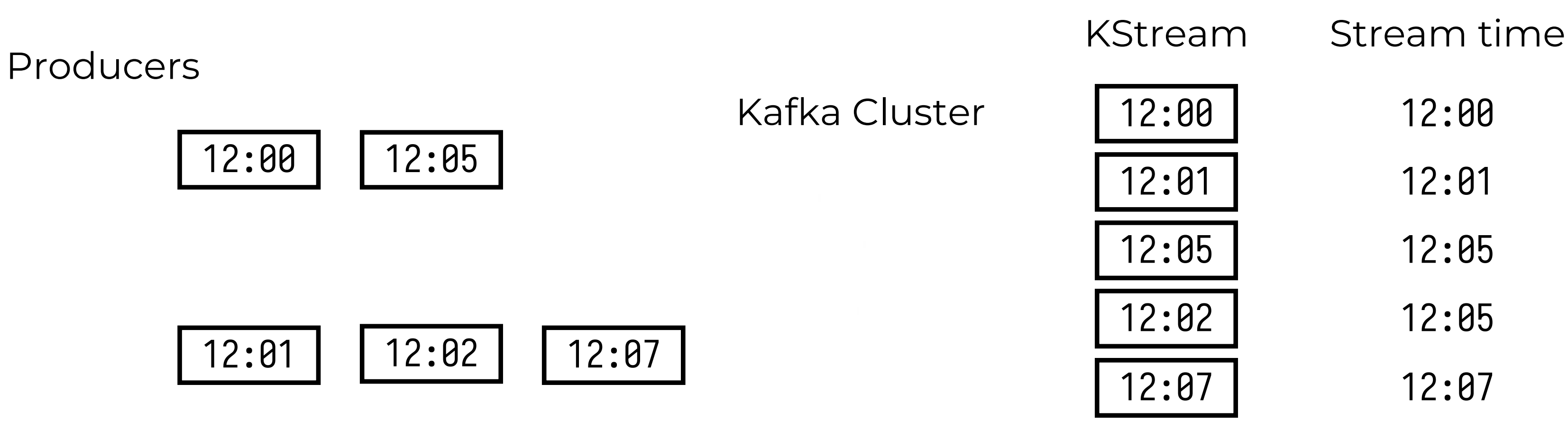
- Event time: time that event occurs (producer)
- Ingestion time: time that Kafka cluster receives event (cluster)
- Stream time: highest timestamp of events seen by Kafka Streams so far

---

You can include a custom timestamp as part of payload (e.g. sensor timestamp attached to event before sending to Kafka producer) and use a custom timestamp extractor

TIMESTAMPS

OUT-OF-ORDERNESS



Timestamp attached to event progresses stream time

Time is important because Kafka is a **distributed system**; **events may arrive out-of-order!**

TIMESTAMPS

GRACE PERIODS

In time-sensitive operations, out-of-order events must be handled

- Set a grace period (maximum expected reasonable delay)
- Events within the grace period will still be processed
- Events out of grace period are **late** and not processed

WINDOWING

TIME-BASED WINDOWING

In general, aggregation on entire stream is impossible as stream is unbounded; two approaches:

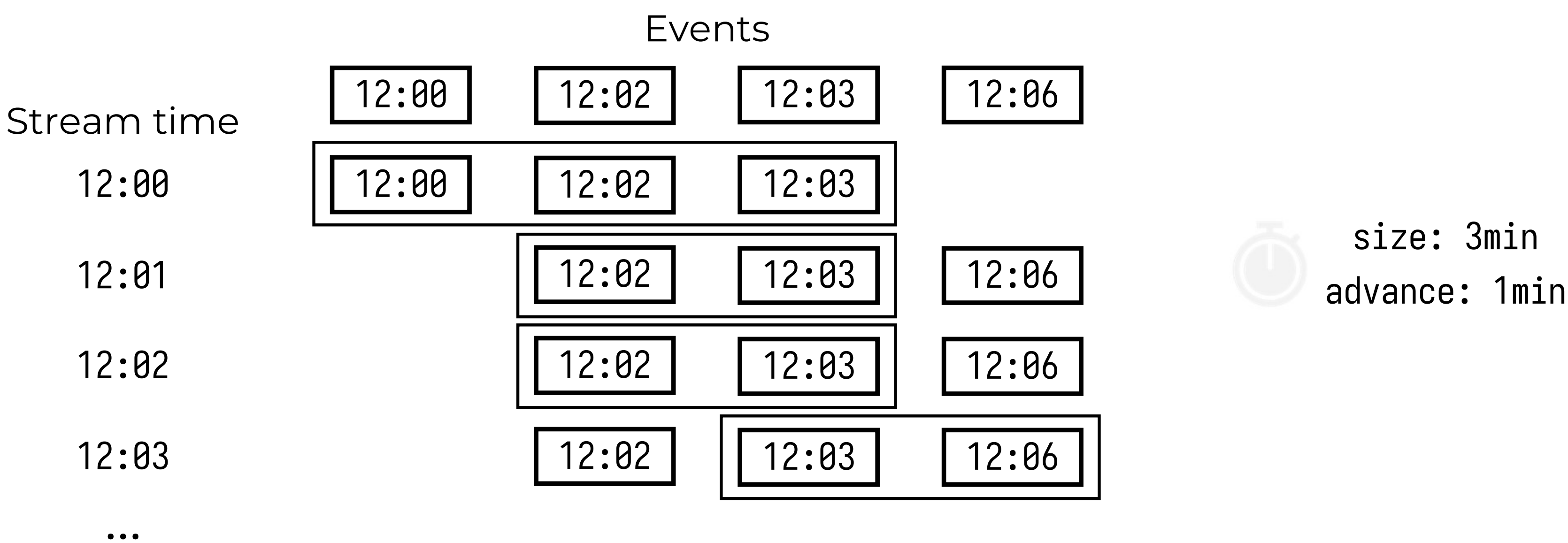
- Aggregation by event (aggregate)
- Aggregation by time (windowing)

Four kinds of windows:

- Hopping
- Tumbling
- Sliding
- Session

WINDOWING

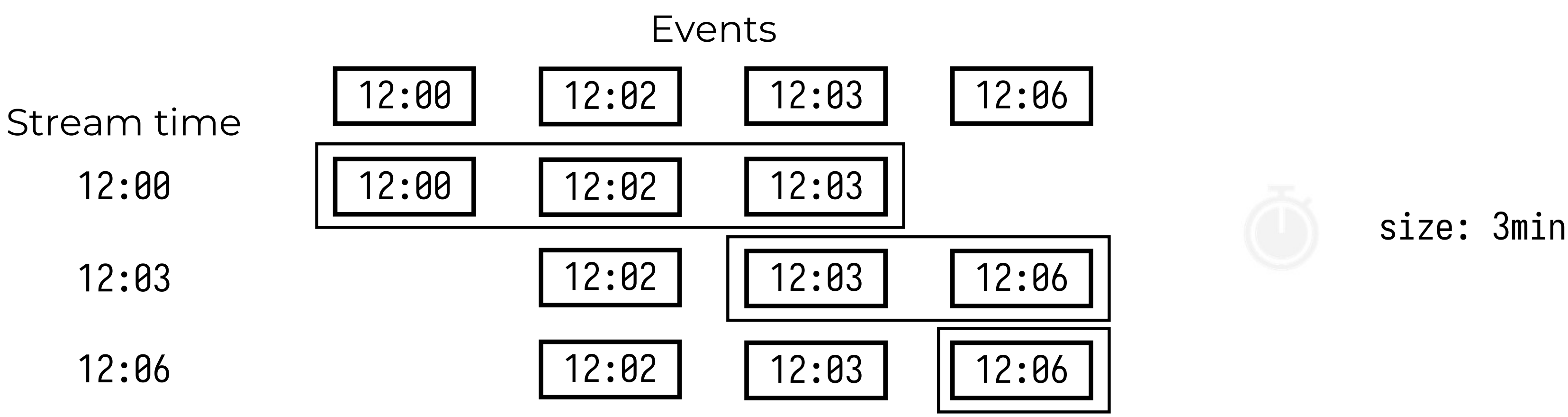
HOPPING WINDOWS



(Potentially) overlapping windows defined by window size (time) and advance time (time step); driven by **time**

WINDOWING

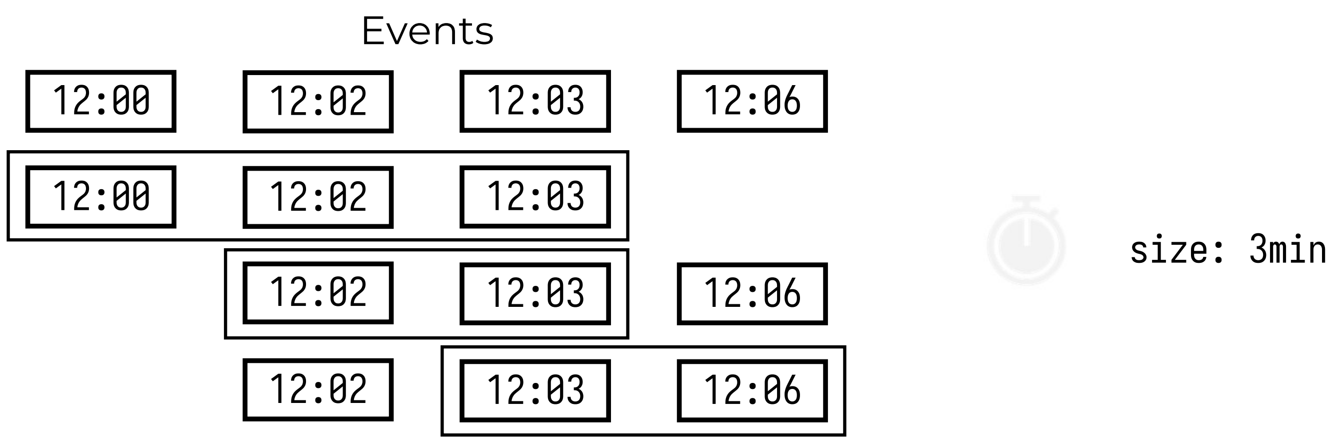
TUMBLING WINDOWS



Hopping window where size and advance time is the same

WINDOWING

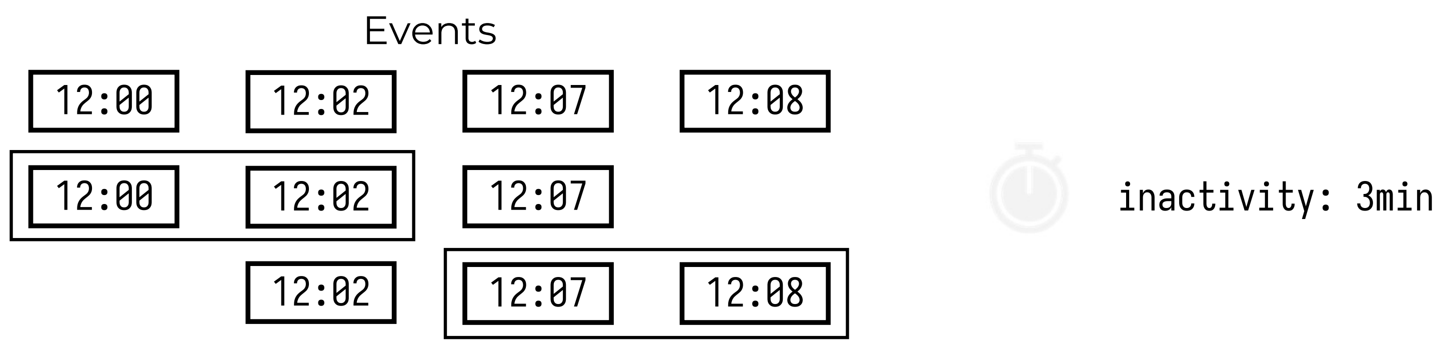
SLIDING WINDOWS



Similar to hopping window except windows are emitted as events are processed (driven by **events**, not time)

WINDOWING

SESSION WINDOWS



Windows are separated by inactivity; useful for analyzing session activity—define time period of inactivity before new session is created

WINDOWING

```
Serde<SumCount> sumCountSerde = Serdes.serdeFrom(new SumCountSerializer(),
        new SumCountDeserializer());
KStream<Integer, Double> source = builder.stream(sourceTopic,
        Consumed.with(Serdes.Integer(), Serdes.Double()));
KGroupedStream<Integer, Double> groupedSource = source.groupByKey();
SlidingWindows window = SlidingWindows
        .ofTimeDifferenceAndGrace(Duration.ofSeconds(5), Duration.ofSeconds(2));
TimeWindowedKStream<Integer, Double> windowedSource = groupedSource.windowedBy(window);
KTable<Windowed<Integer>, SumCount> averager = windowedSource
        .aggregate(SumCount::new,
            (k, v, s) -> s.put(v),
            Materialized.with(Serdes.Integer(), sumCountSerde))
        .suppress(Suppressed.untilWindowCloses(Suppressed.BufferConfig.unbounded()));
KStream<Windowed<Integer>, Double> windowedAverages = averager.toStream()
        .mapValues(SumCount::get)
        .peek((k, v) -> System.out.printf("%s: %.2f\n", k.window(), v));
KStream<Integer, Double> averages = windowedAverages.map((k, v) -> KeyValue.pair(k.key(), v));
averages.to(sinkTopic, Produced.with(Serdes.Integer(), Serdes.Double()));
KafkaStreams s = new KafkaStreams(builder.build(), p);
s.start();
```

Now we can perform windowed aggregations!

KEY POINT #3

Time is important in event stream processing!

JOINS

JOINS

Just like with `Flux` and `Mono`, we would like to merge events from different nodes in our topology:

- Merging separate processor results
- Merging events from two source topics

`KStreams` and `KTables` both support **joins**

- Stream-Stream joins (windowed)
- Stream-Table joins
- Table-Table joins

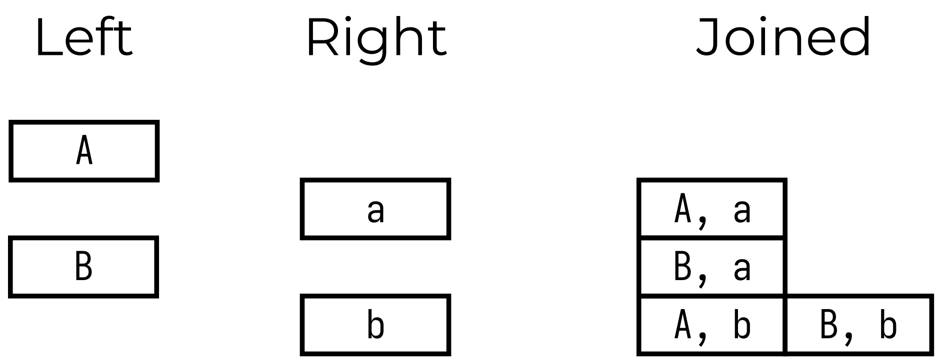
Joined records have the same key

JOINS

JOIN TYPES

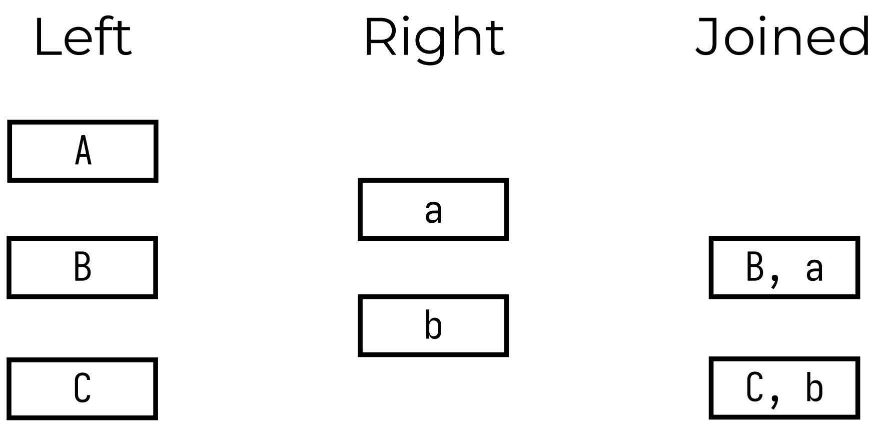
- **Inner join**: join records of same key only when record exists in both sides
- **Left outer join**: join records of same key, always emit records of left side (right side might be null)
- **Outer join**: join records of same key, always emit records even if one side is null

STREAM-STREAM JOINS



- Records from both sides are kept in windows
- Once event from either side arrives, pair with all records in other side within window

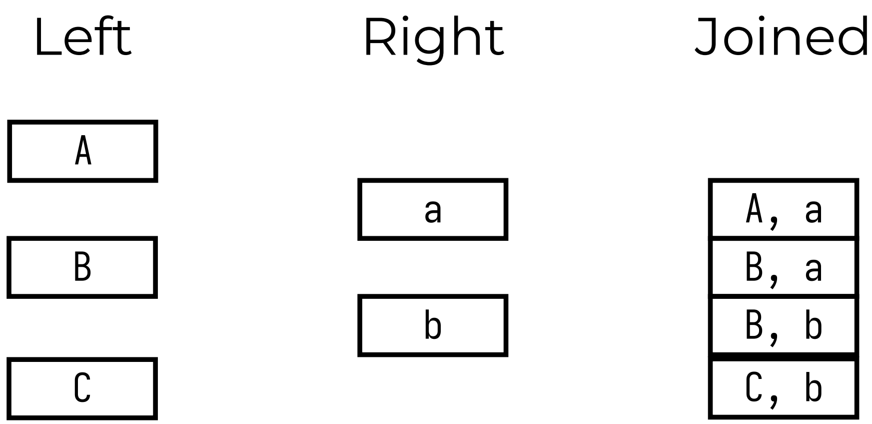
STREAM-TABLE JOINS



- Left (`KStream`) drives join
- When event in stream arrives, look up latest value in `KTable`
- If right is `KTable`, table updates are only performed by time

In this example, depending on join type, the first emission of `A` in the `KStream` may emit join result `(A, null)`

TABLE-TABLE JOINS



Joined table reflects latest state of both sides

In this example, depending on join type, the first emission of `A` in the `KStream` may emit join result `(A, null)`

JOINS

```
Serde<User> userSerde = Serdes.serdeFrom(new UserSerializer(),
        new UserDeserializer());

KTable<Integer, String> names = builder.table(namesTopic,
        Consumed.with(Serdes.Integer(), Serdes.String()));
KStream<Integer, Double> balances = builder.stream(balancesTopic,
        Consumed.with(Serdes.Integer(), Serdes.Double()));

KTable<Integer, Double> aggregatedBalances = balances.groupByKey()
        .reduce(Double::sum);

KTable<Integer, User> joinedTable = names.join(aggregatedBalances,
        (x, y) -> User.empty(-1).ofName(x).ofAccountBalance(y),
        Materialized.with(Serdes.Integer(), userSerde));

joinedTable.toStream().map((k, v) -> KeyValue.pair(k, v.ofId(k)))
        .peek((k, v) -> System.out.println(v))
        .to(sinkTopic, Produced.with(Serdes.Integer(), userSerde));
```

Now we can actually use Kafka to reflect updates to user state changes!

NOTES ON JOINS

CO-PARTITIONING

Because Kafka Streams works in parallel, two topics must have same number of partitions and same partitioning strategy!

Otherwise, possible for key to be in partition 0 in one topic and in partition 1 in another topic, records are lost!

KEY POINT #4

Merge two streams/tables with joins!

CONTENTS

- Event Stream Processing with Kafka
- Kafka Streams
- Stream-Table Duality
- Time
- Joins

KEY POINTS

- Use the Kafka Streams API to process event streams on Kafka!
- Use KTables for fault-tolerant and scalable stateful stream processing!
- Time is important in event stream processing!
- Merge two streams/tables with joins!

IT5100B PROJECT

IT5100B PROJECT

ADMINISTRIVIA & OBJECTIVES

40% of overall grade, due one week after course conclusion (8 Mar)

Goals:

- Tie up concepts from entire course into hands-on project
- Practice writing good code
- Practice configuring architecture and designing system according to requirements
- Have fun!

IT5100B PROJECT

FEATURES

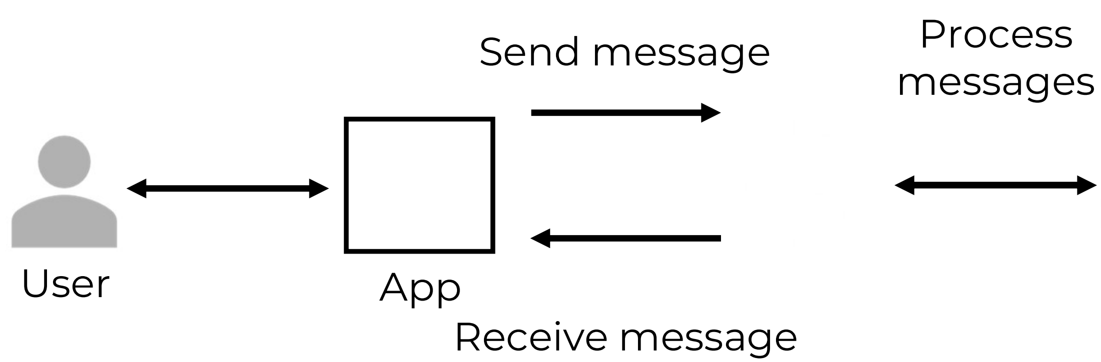
Few features, many components

- Reactor Kafka as mock front-end to interact with event bus
- Apache Kafka cluster as event bus
- Apache Flink as Stream Processor

IT5100B PROJECT

DESCRIPTION

You will be writing a simple console-based chat application from scratch!



IT5100B PROJECT

GRADING

Base grade (30%): perfect code and configuration, console application, one message processor (censoring profanities)

Full grade (40%): fulfilled requirements for base grade and add one more feature of your choice (message rate limiter? user aliases?)

Additional features incorporating technologies not taught in IT5100B is prohibited (Spring WebFlux, any web front-end etc.)*

*Use it for your own GitHub portfolio, don't let me steal it :P

IT5100B PROJECT

DELIVERABLES

- Console application and Flink job source code
- Report, one page or so, point-form is okay, keep it short and sweet!

Do not use any exotic docker images, you will not submit them

IT5100B PROJECT

Formal project instructions will be released shortly, but you can start now (there will be no provided template files, the configuration is all up to you)

Have fun!!