

# 실시간 빅데이터 처리 spark/flink - Part4 스트림 프로세싱 – 아파치 플링크

# 이 수업에서 다루고자 하는 것

## 이론

- Stateful / Timely 스트리밍 처리
- Architecture & Code structure
- Finite / Unfinite data
- Operator
- Keyed State
- Checkpointing / Savepoint
- Event Processing(CEP)
- Flink ML

# 이 수업에서 다루고자 하는 것

## 실습

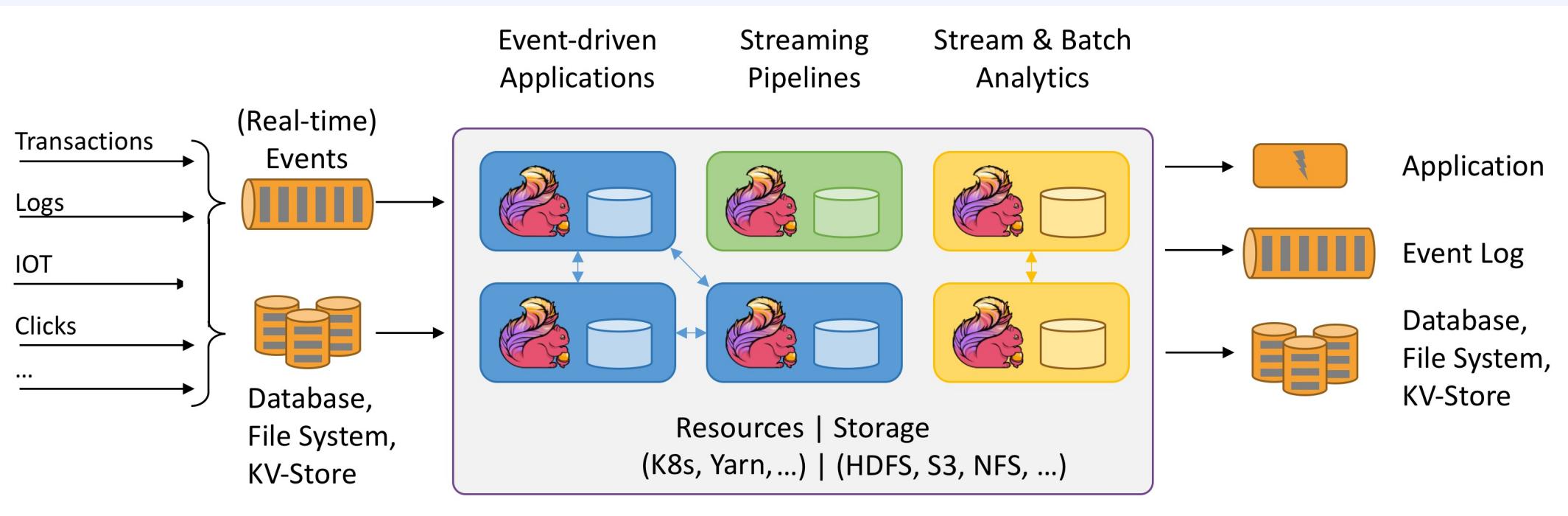
- Source & Sink
- Operator
- Keyed State
- Event Processing(CEP)
- Flink ML

# Chapter 1. Abstraction

# Chapter 1.

## 01. 다른 Streaming Tool과의 비교

# 01. Streaming Data란



## 01. Streaming Data에서 해결해야 할 지점

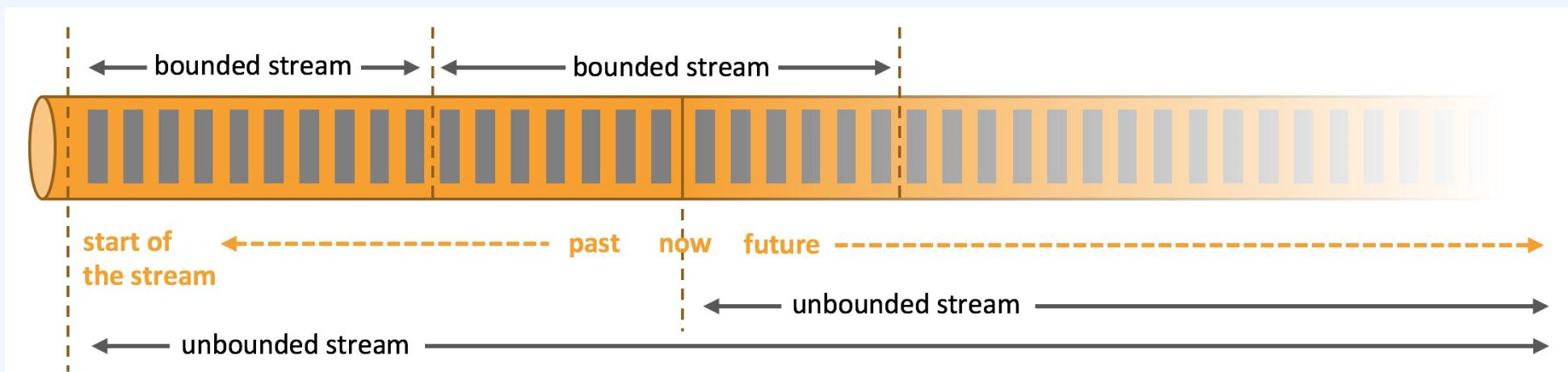
- Out-of-order
- Late events
- Stateful processing
- Real-time processing
- Fault tolerance
- Scalability
- Integration with other systems

# 01. 다른 Streaming Tool과 비교

Feature	Apache Flink	Spark Streaming	Kafka Streams
Stream processing	True	Mini-batch	True
Latency	Low	Higher	Low
Event time	Advanced support	Less mature support	Limited support
Windowing	Flexible and extensive	Limited capabilities	Less flexible
State management	Built-in and efficient	External data store	Local state storage
Fault tolerance	Strong guarantees	Strong guarantees	Exactly-once
Memory management	Fine-grained control	Unified memory model	Efficient
Integration	Rich ecosystem	Unified ecosystem	Tight integration
Learning curve	Steeper	Easier due to Spark	Lightweight Library
Scalability	High throughput	Scalable	Less scalable

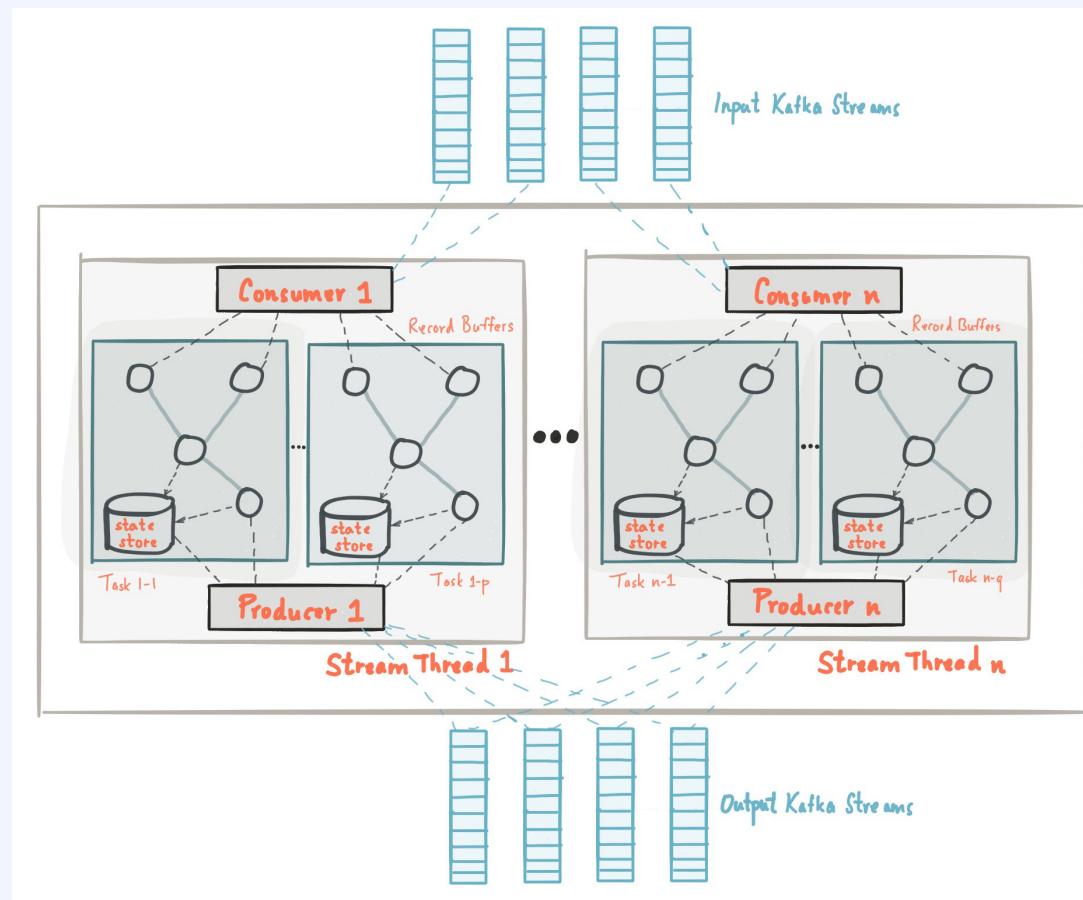
# 01. Flink: Fourth-Generation

- Micro-batch
  - Spark: ‘Streaming is a special case’
  - Flink: ‘Batch is a special case’



# 01. Flink: Fourth-Generation

- Sub-topology



## 01. 다른 Streaming Tool의 장점

- Spark Streaming
  - 배치 처리와 동일한 API
  - Spark Ecosystem과의 통합
- Kafka Streams
  - Lightweight & Simple API
  - Kafka와의 강력한 통합

# 01. Streaming Processing

- Timely Stream Processing
  - Out-of-order
  - Late events
- Stateful Stream Processing
  - Stateful processing

## 01. 예제: Spark Streaming

```
val filteredStocks = stockStream  
    .map(Stock.fromCSV)  
    .filter(_.price > 100.0)
```

## 01. 예제: Kafka Streams

```
KStream<String, Stock> filteredStocks = stockStream  
    .mapValues(Stock::fromCSV)  
    .filter((key, value) -> value.getPrice() > 100.0);
```

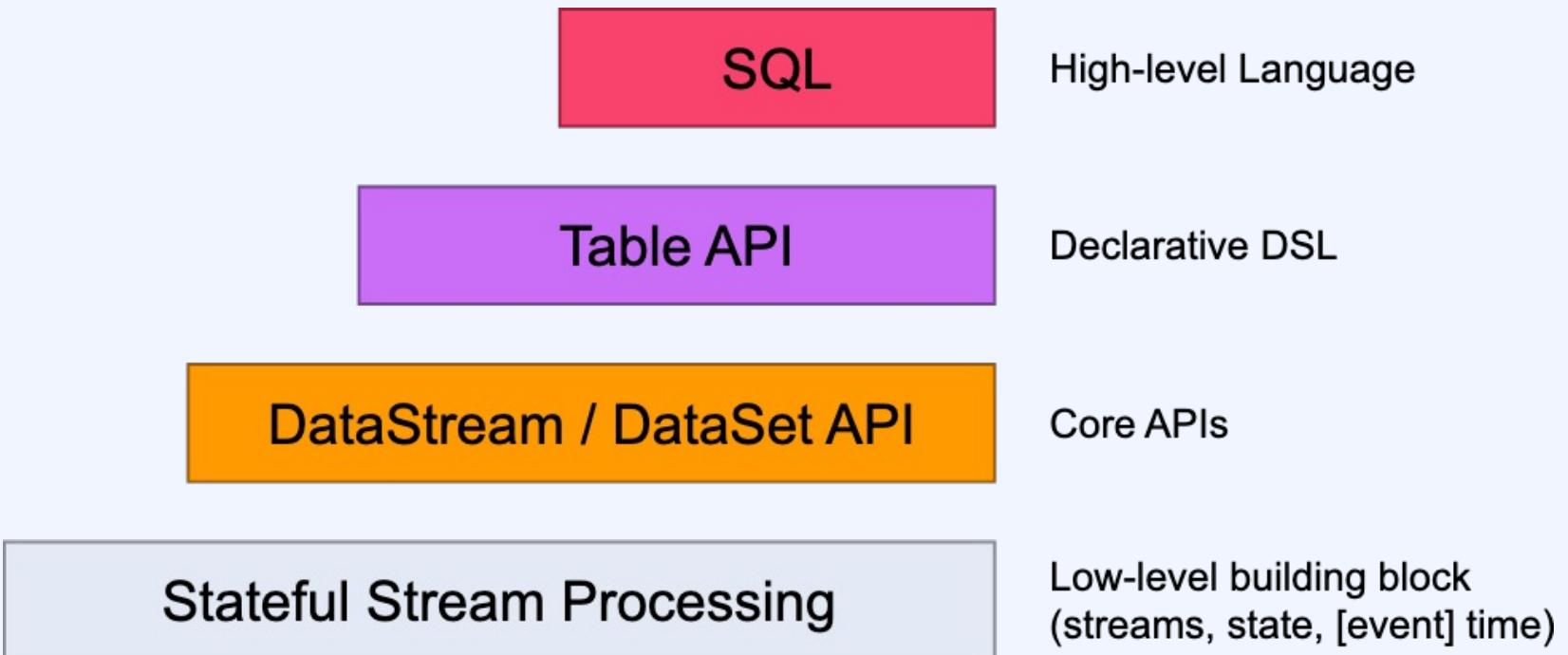
## 01. 예제: Apache Flink

```
DataStream<Stock> filteredStocks = stockStream  
    .map(Stock::fromCSV)  
    .filter(stock -> stock.getPrice() > 100.0);
```

# Chapter 1.

## 02. 고수준 / 저수준 API

## 02. Flink's APIs



## 02. 예제를 통한 비교 (High Level API - map())

```
DataStream<Integer> result = input.map(i -> i * 2)
```

---

```
@Public  
@FunctionalInterface  
public interface MapFunction<T, O> extends Function, Serializable {  
    O map(T value) throws Exception;  
}
```

## 02. 예제를 통한 비교 (Low Level API – ProcessFunction{})

```
public class MapProcessFunction extends ProcessFunction<Integer, Integer> {  
    @Override  
    public void processElement(Integer value, Context ctx, Collector<Integer> out)  
        throws Exception {  
        out.collect(value * 2); // Simple map operation - doubling the input value  
    }  
}
```

```
DataStream<Integer> result = input.process(new MapProcessFunction());
```

## 02. 예제를 통한 비교 (High Level API - reduce())

```
DataStream<Integer> result = input.keyBy(i -> 0).reduce((v1, v2) -> v1 + v2);
```

```
DataStream<Integer> result = input.keyBy(i -> 0).reduce(Integer::sum);
```

## 02. 예제를 통한 비교 (Low Level API – ProcessFunction{})

```
public class StatefulReduceFunction extends KeyedProcessFunction<Integer,  
Integer, Integer> {  
    @Override public void processElement(Integer value, Context ctx,  
Collector<Integer> out) throws Exception {  
    Integer currentSum = sumState.value();  
    sumState.update(currentSum + value);  
    out.collect(sumState.value());  
}  
}
```

## 02. 예제를 통한 비교 (Low Level API – ProcessFunction{})

```
public class StatefulReduceFunction
    extends KeyedProcessFunction<Integer, Integer, Integer> {
    private ValueState<Integer> sumState;
    @Override public void open(Configuration parameters) throws Exception {
        ValueStateDescriptor<Integer> descriptor =
            new ValueStateDescriptor<>("sum", // state name
                TypeInformation.of(new TypeHint<Integer>() {}), // type
                information
                0); // default value of the state, if nothing else is set
        sumState = getRuntimeContext().getState(descriptor);
    }
}
```

## 02. 고수준 및 저수준 API 비교

- High Level API
  - streams나 batches 데이터를 다루기 위한 추상 데이터 타입 제공  
-> DataStreams 및 DataSets
  - 대부분의 사용 사례에 적합하며, 사용하기 쉬운 인터페이스 제공  
-> map, flatMap, filter, window, reduce, join, coGroup
- Low Level API
  - Stream processing에 대한 더 많은 제어 제공
    - Stateful 및 Timely stream processing

# Chapter 1.

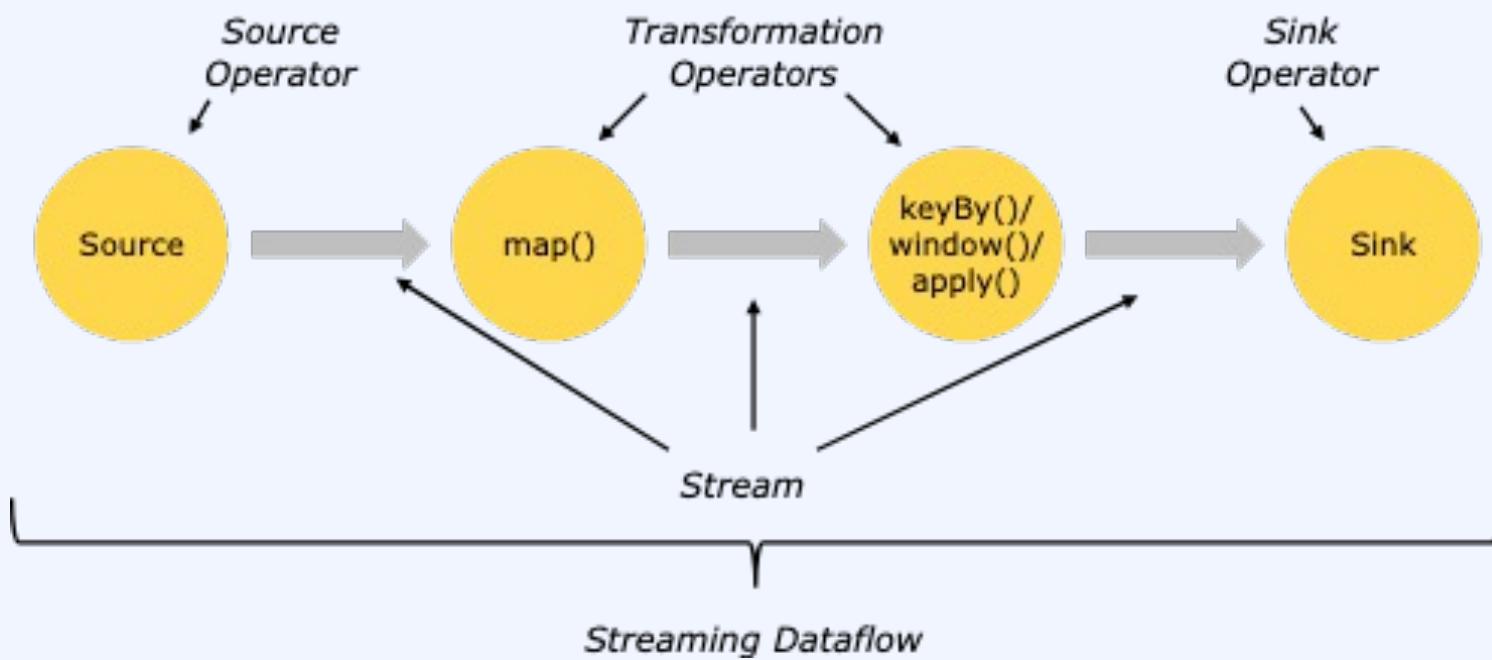
## 03. Stateful 스트리밍 처리

## 03. Flink에서의 State

- Keyed State
- Operator State
  - Broadcast State

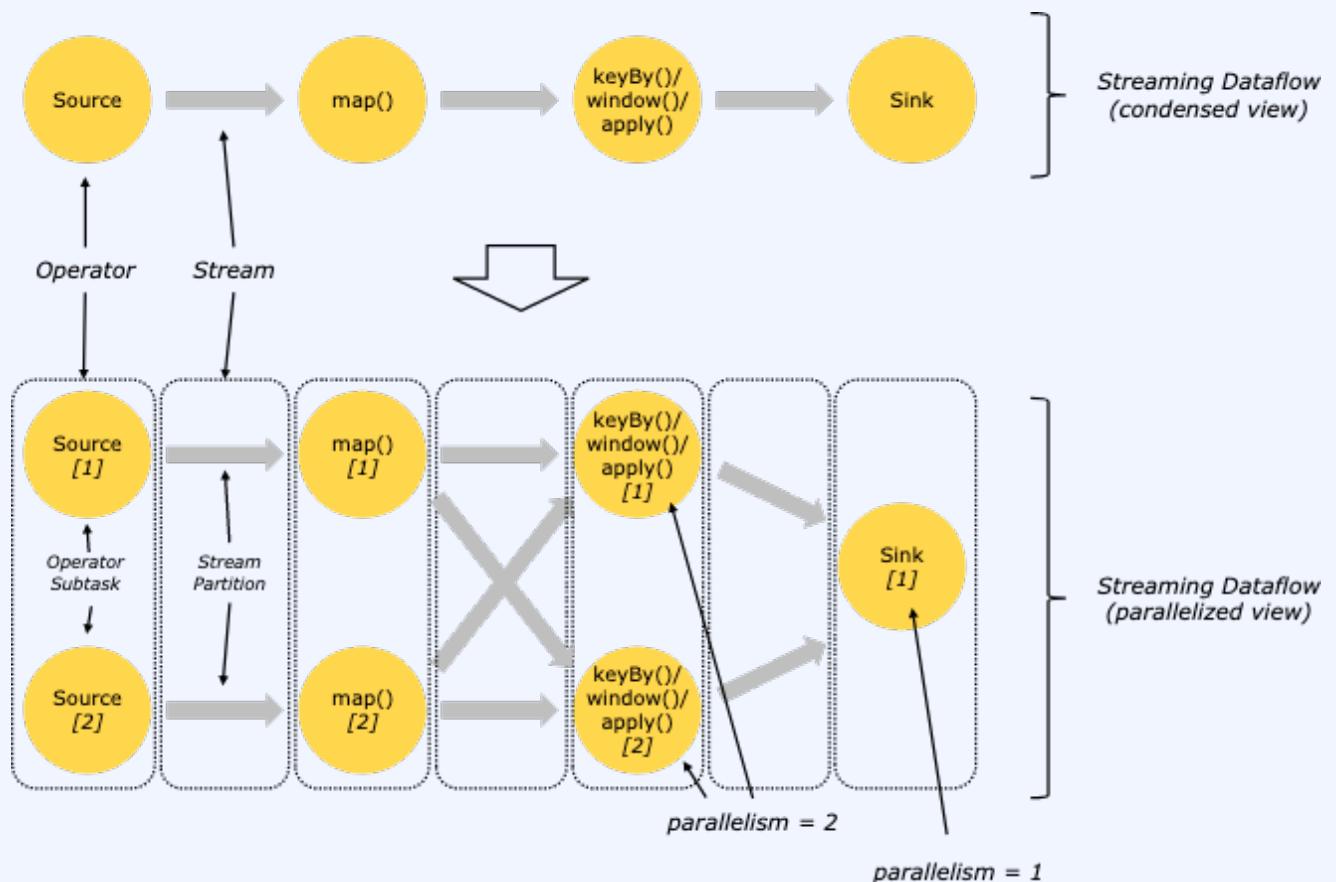
## 03. Operator State

- State on a per-operator basis



## 03. Operator State

- Operator state is bound to the operator and its parallelism.



### 03. 예제를 통한 확인 (RichParallelSourceFunction{})

```
public class MyKafkaSource
    extends RichParallelSourceFunction<String> {
    private ListState<Long> offsetState;

    @Override public void open(Configuration parameters) throws Exception {
        ListStateDescriptor<Long> descriptor =
            new ListStateDescriptor<>("offsets", // state name
                TypeInformation.of(new TypeHint<Long>() {}));
        information
        offsetState = getRuntimeContext().getListState(descriptor); }
    ...
}
```

## 03. Identify by Key

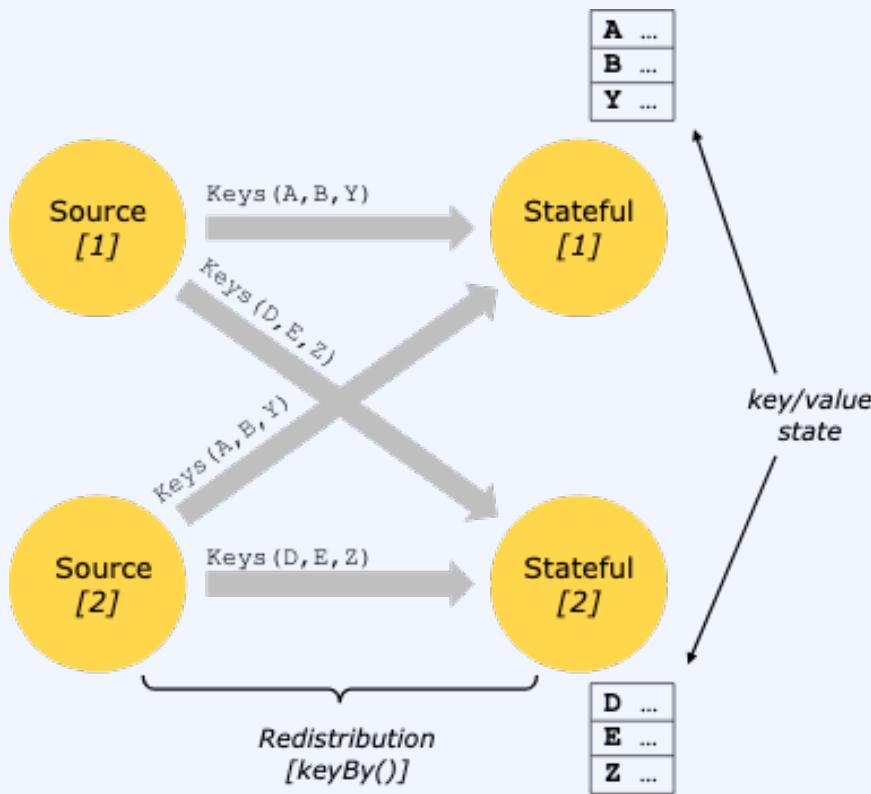
- Identifier
  - Operator는 OperatorID
  - State는 Keyed State

## 03. Stateful Operation on Same Key

- State는 Keyed State, Keyed State는 State
- Key는 들어오는 streams을 가상으로 partitioning 방식

## 03. Keyed State

- Key Groups are the atomic unit (to redistribute Keyed State)



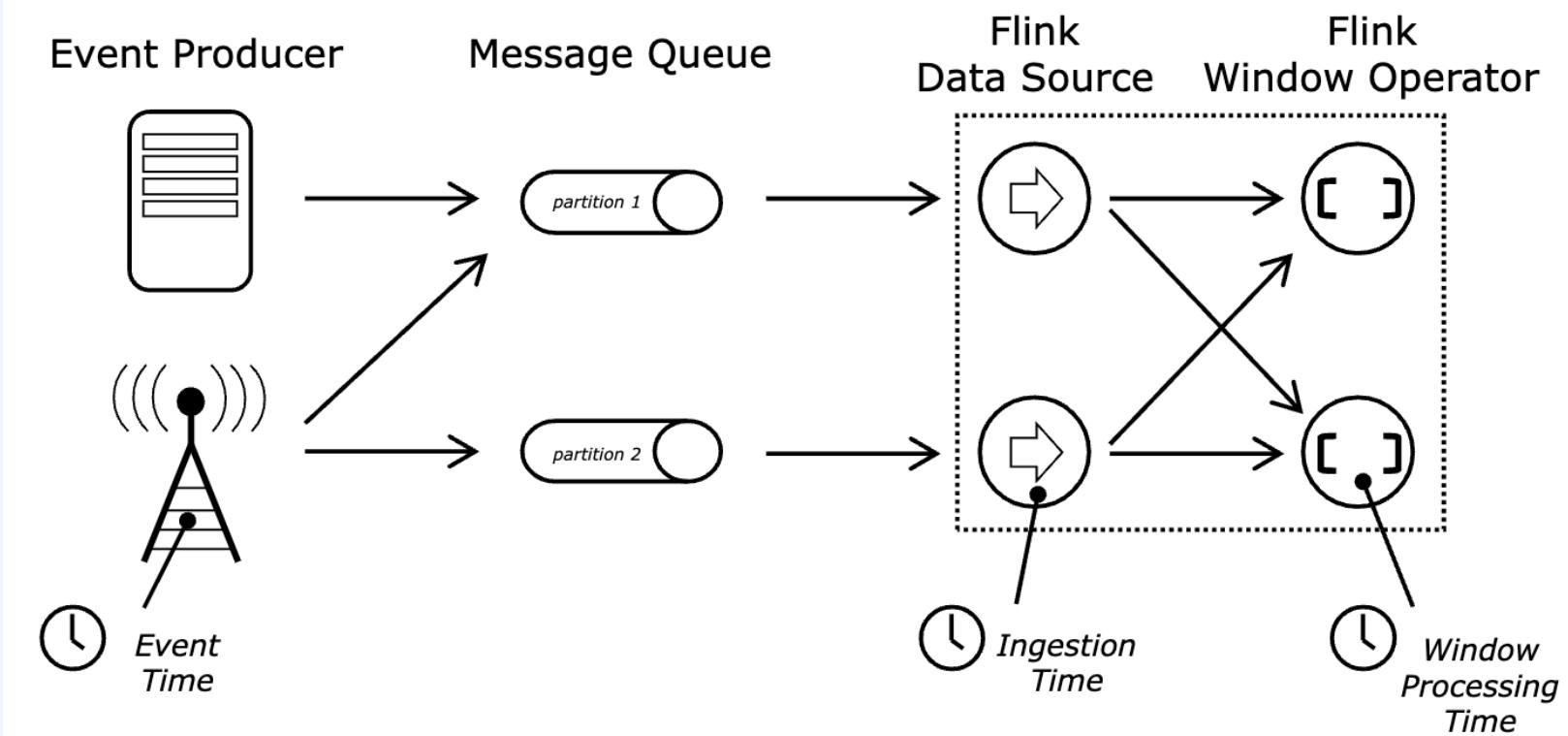
# Chapter 1.

## 04. Timely 스트리밍 처리

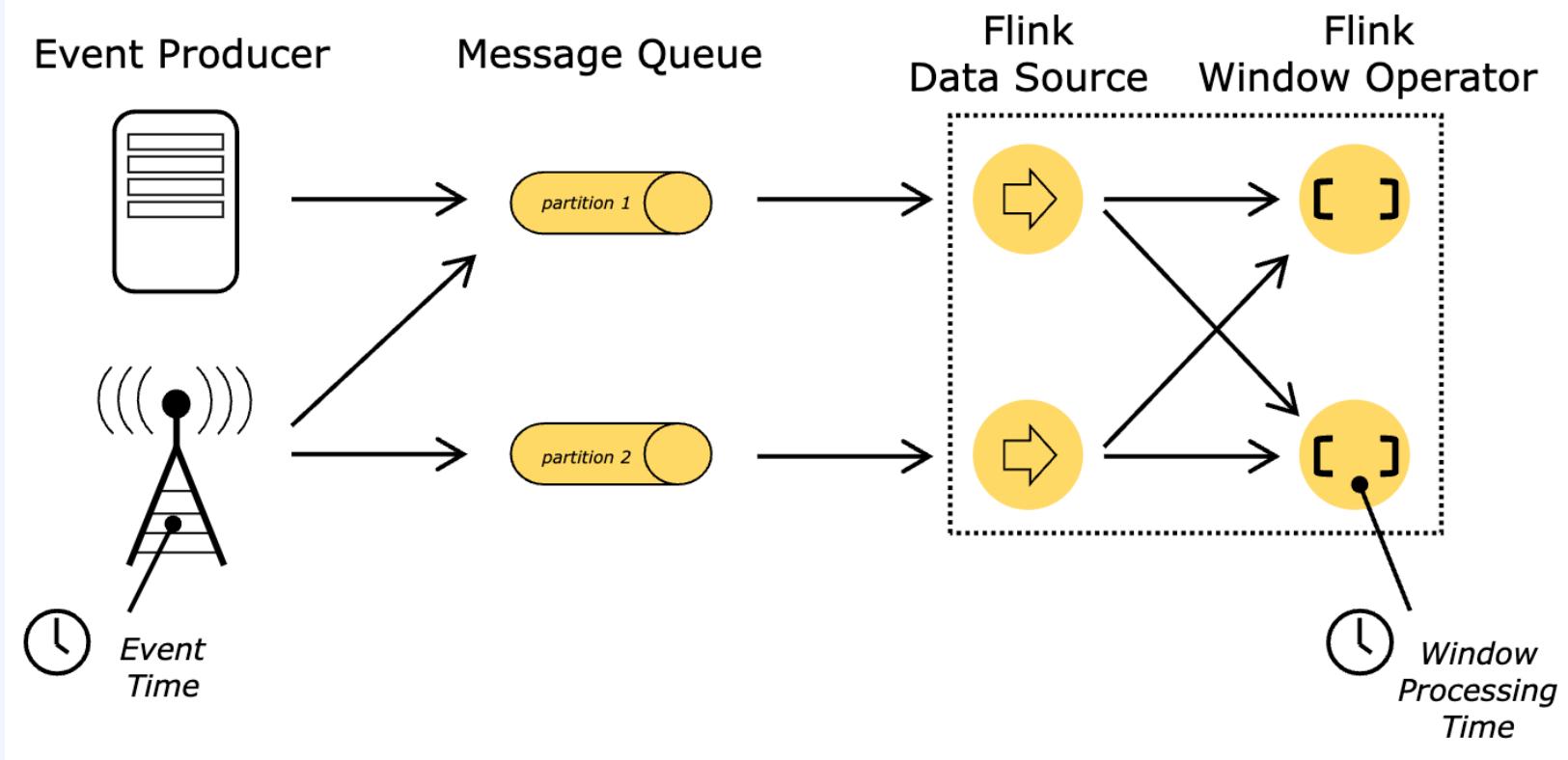
## 04. Flink에서 시간을 결정하는 법

- Event Time
- Processing Time
- Ingestion Time(1.12 버전 이후 deprecated)

## 04. Ingestion Time(Past)



## 04. Ingestion Time(Current)



## 04. Point of Time

- Timely Stream Processing
- Processing Time

## 04. Processing Time

- Processing Time이 10:00 시점에 데이터 처리를 진행하도록 지시
  - 특정 시점( $T$ )
    - 1번 노드(9:59): 처리하지 않음
    - 2번 노드(10:00): 처리 진행
  - 특정 시점 1분 후( $T+1$ )
    - 1번 노드(10:00): 처리 진행
    - 2번 노드(10:01): 기존에 처리함

## 04. Event Time

- Internal Event Time Clock (vs Wall Clock)
  - Updated by Watermark

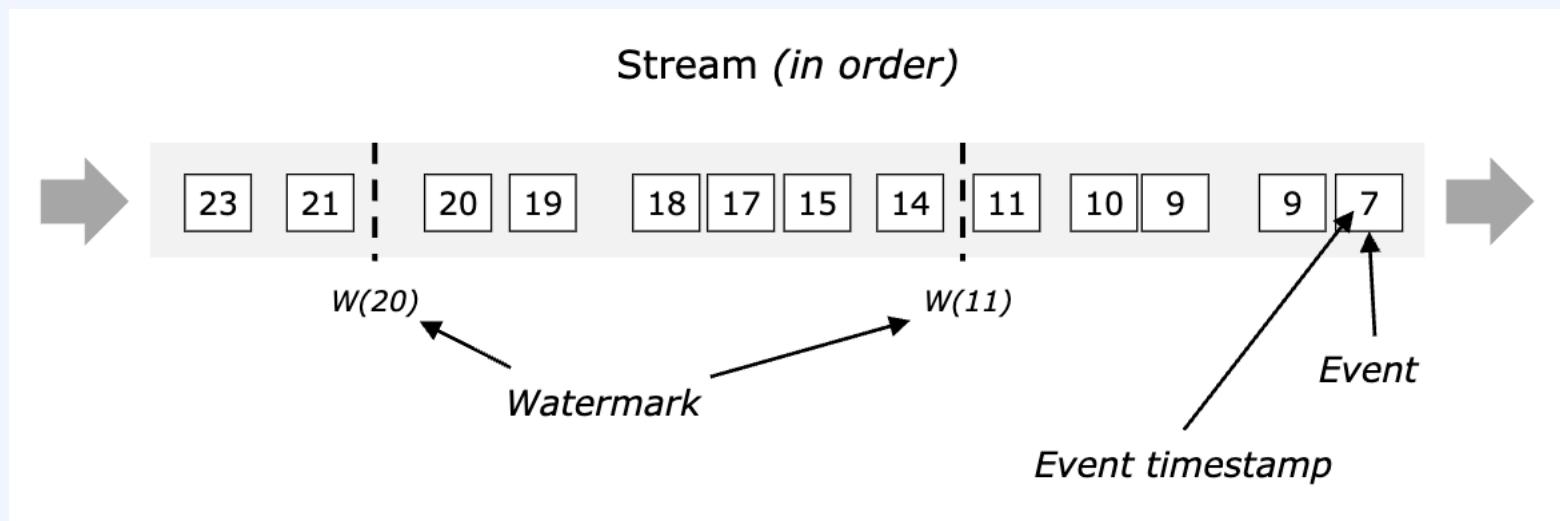
## 03. Watermark & Timestamp

```
DataStream<Event> inputStream = ...;
```

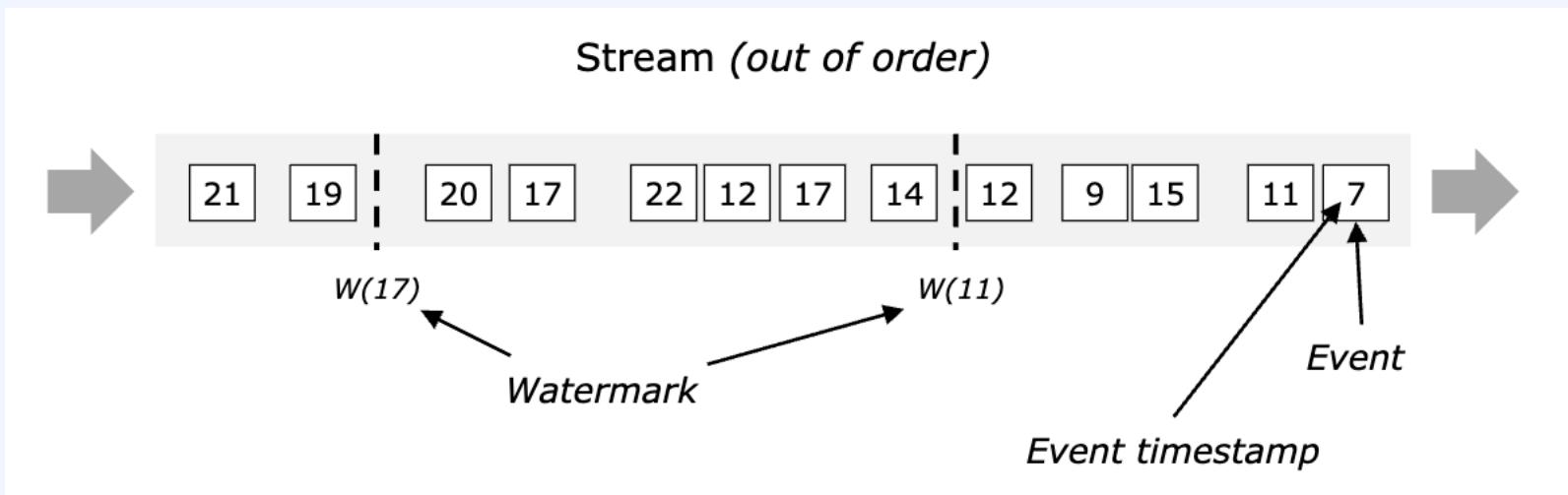
```
WatermarkStrategy<Event> wmStrategy = WatermarkStrategy  
    .<Event>forMonotonousTimestamps()  
    .withTimestampAssigner((event, timestamp) -> event.getCreationTime());
```

```
DataStream<Event> withTimestampsAndWatermarks =  
inputStream.assignTimestampsAndWatermarks(wmStrategy);
```

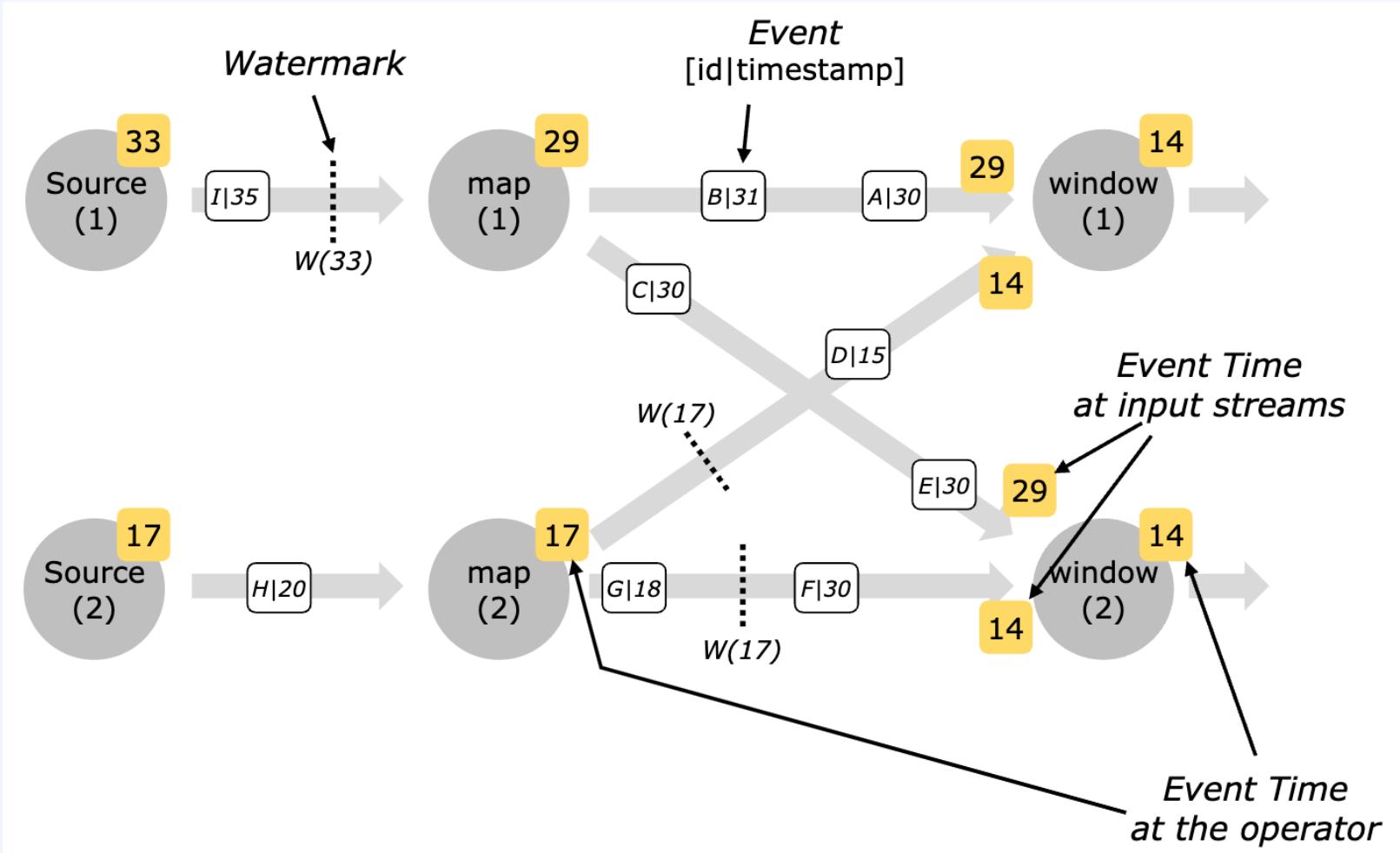
## 04. Out-of-order



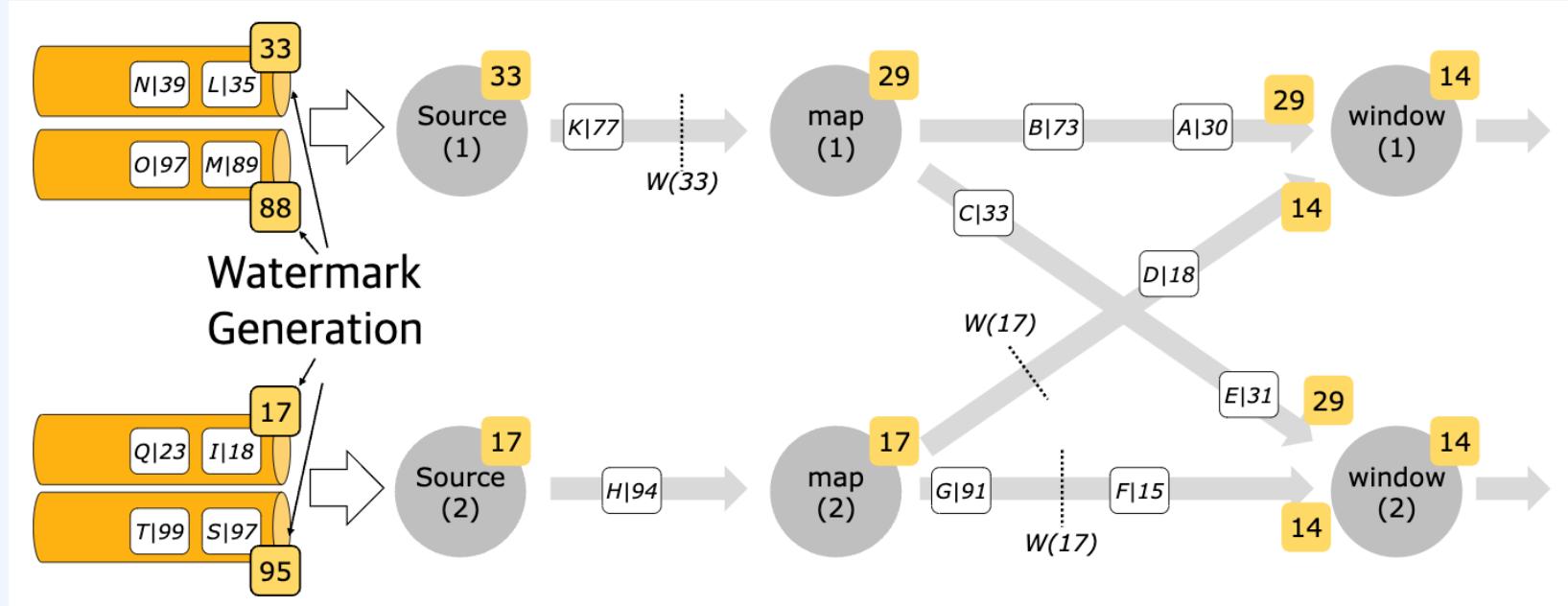
## 04. Out-of-order



## 04. Watermarks in Parallel Streams



## 04. Watermarks in Parallel Streams



## 04. Lateness

```
DataStream<Event> inputStream = ...;
```

```
WatermarkStrategy<Event> wmStrategy = WatermarkStrategy  
    .<Event>forBoundedOutOfOrderness(Duration.ofSeconds(5))  
    .withTimestampAssigner((event, timestamp) -> event.getCreationTime());
```

```
DataStream<Event> withTimestampsAndWatermarks =  
inputStream.assignTimestampsAndWatermarks(wmStrategy);
```

## 04. forBoundedOutOfOrderness()

- $T - D$  ( ~~$\text{not } T + D$~~ )
  - T: 관찰된 최대 Timestamp
  - D: 허용된 지연
- e.g. T: 10:05, D: 1분  $\rightarrow T - D == 10:04$

# Chapter 1.

## 05. Architecture

## 05. 데이터 처리와 흐름

- A Flink Application is made up of operators and streams

## 05. getExecutionEnvironment()

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

...

```
env.execute()
```

## 05. StreamExecutionEnvironment

- Flink Jobs의 실행
  - Local JVM -> Local(StreamExecution)Environment
  - Remote Cluster -> Remote(StreamExecution)Environment

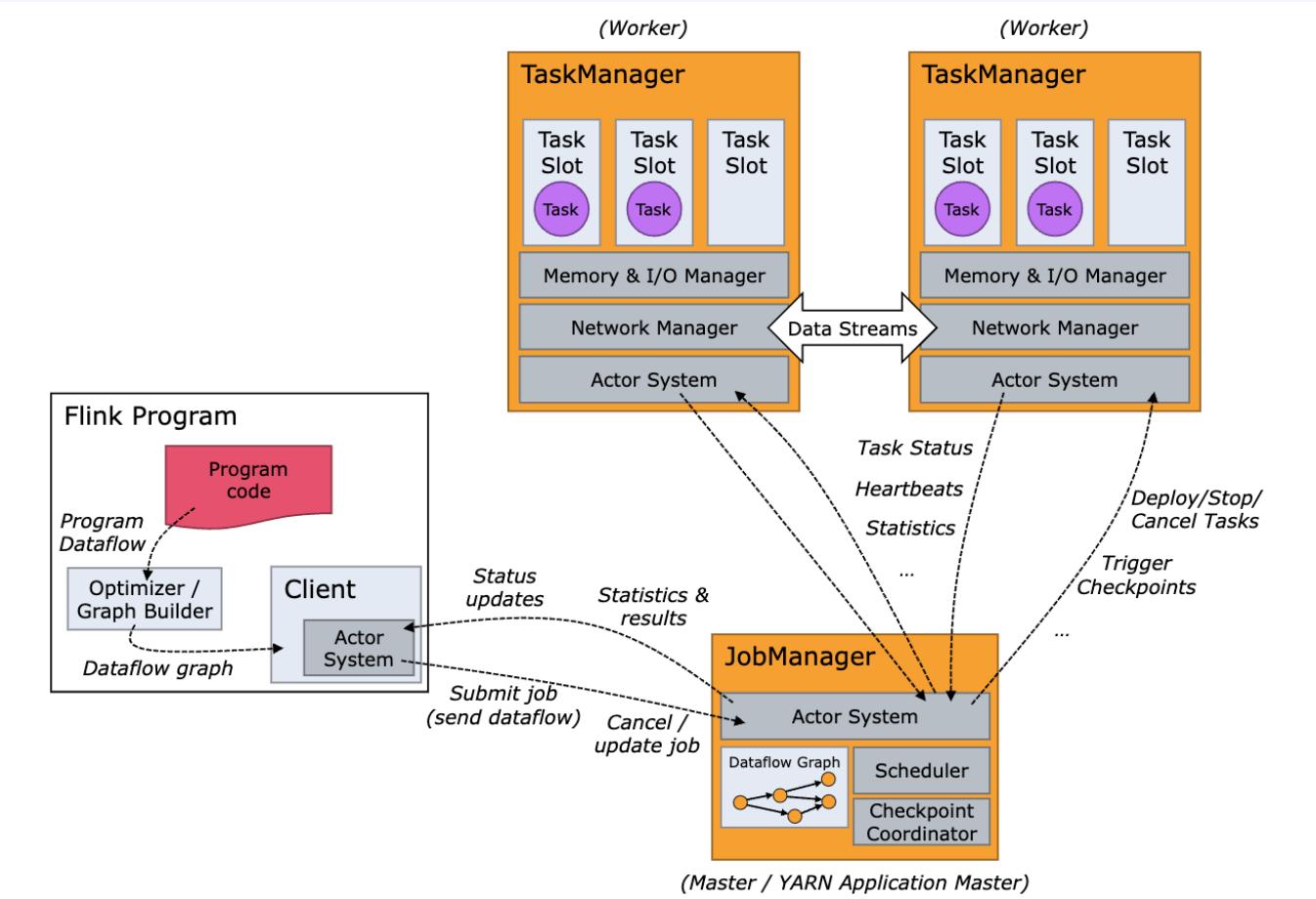
## 05. Anatomy of a Flink Cluster

- (a) JobManager
- (one or more) TaskManagers

## 05. Glossary of a Flink Cluster

- Application -> Job -> Task
- Relation
  - Application is consisting of Jobs
  - Job is composed of Tasks

# 05. Anatomy of a Flink Cluster



## 05. Graph in Flink

- StreamGraph -> JobGraph -> Physical Plan -> ExecutionGraph
- JobGraph
  - JobVertices: Operators in the program
  - JobEdges: Streams between the operators
- ExecutionGraph
  - ExecutionVertices: each parallel Subtask of the job
  - ExecutionEdges: each Stream between the subtasks

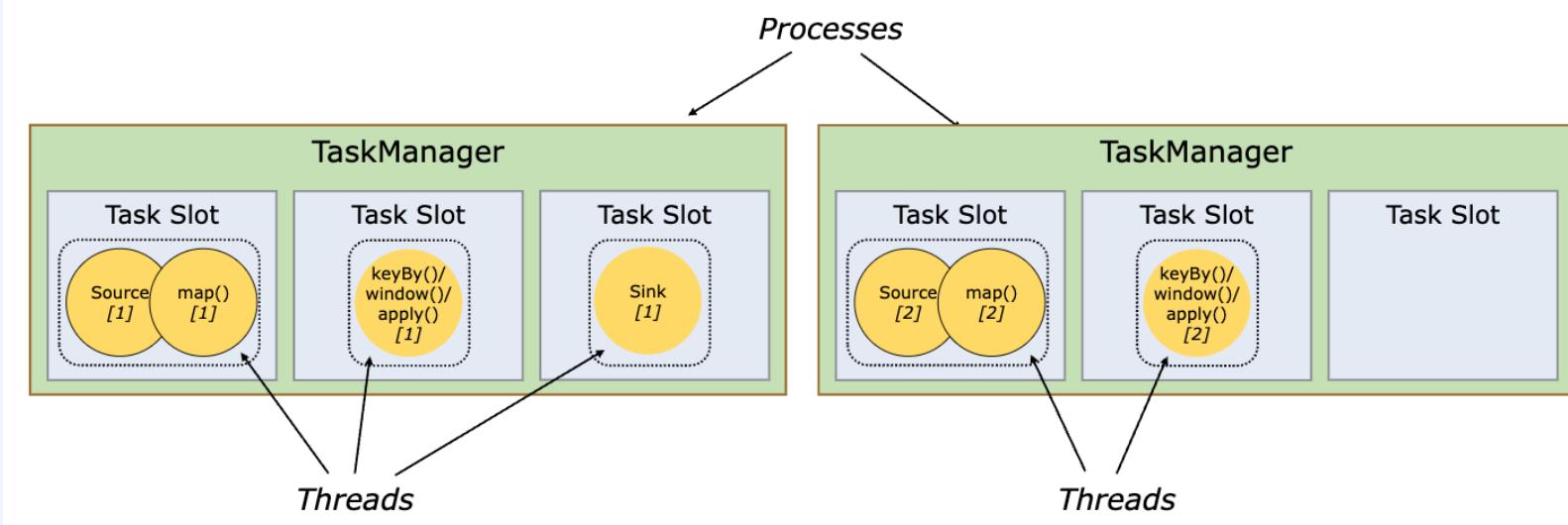
## 05. JobManager

- JobManager
  - ResourceManager
  - Dispatcher
  - JobMaster

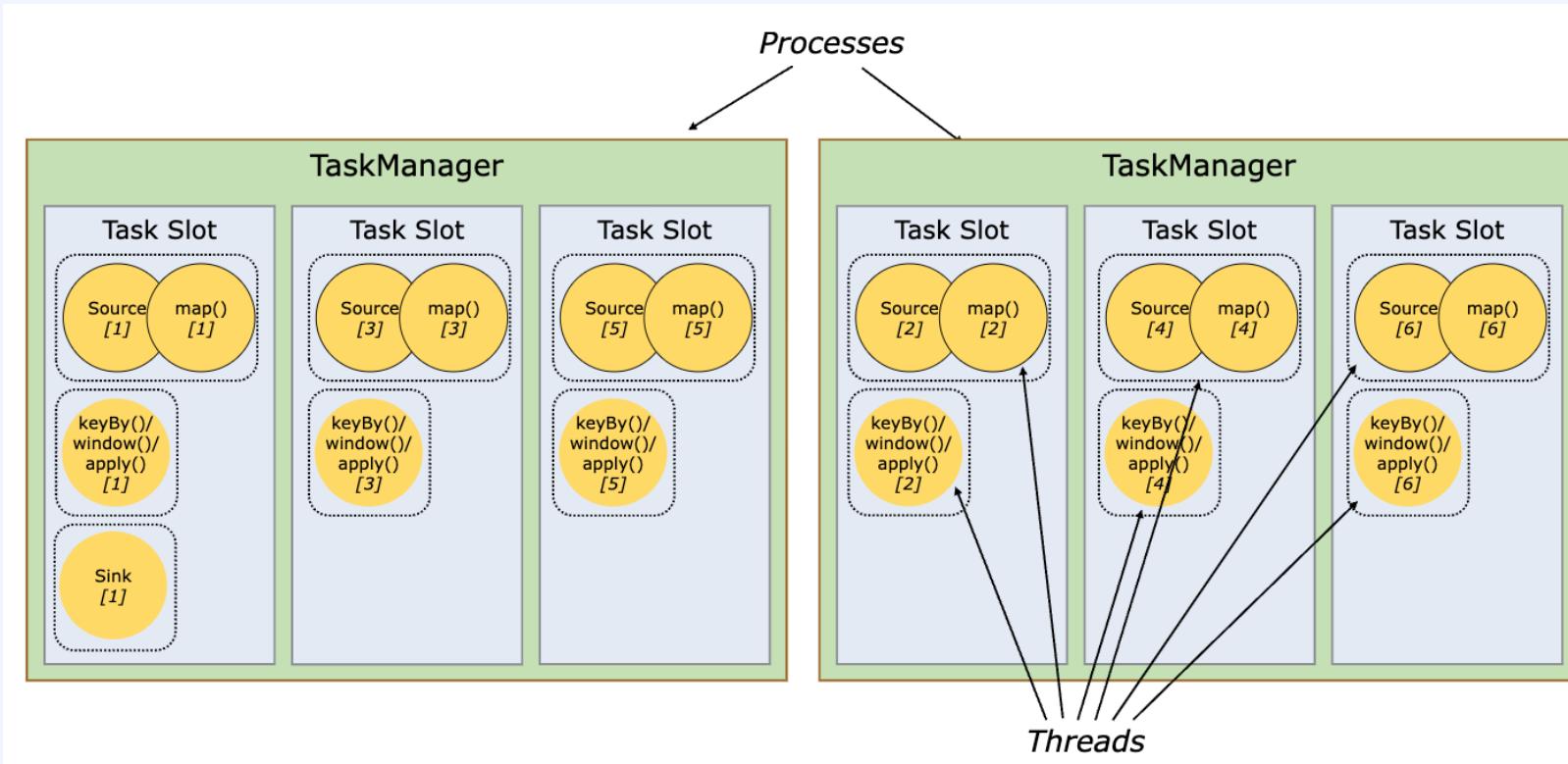
## 05. TaskManager

- 하나의 TaskManager는 독립적으로 실행되는 여러 Task Slot으로 분할
- 또한 Task Slot과 Operator는 일대일 대응이 아님

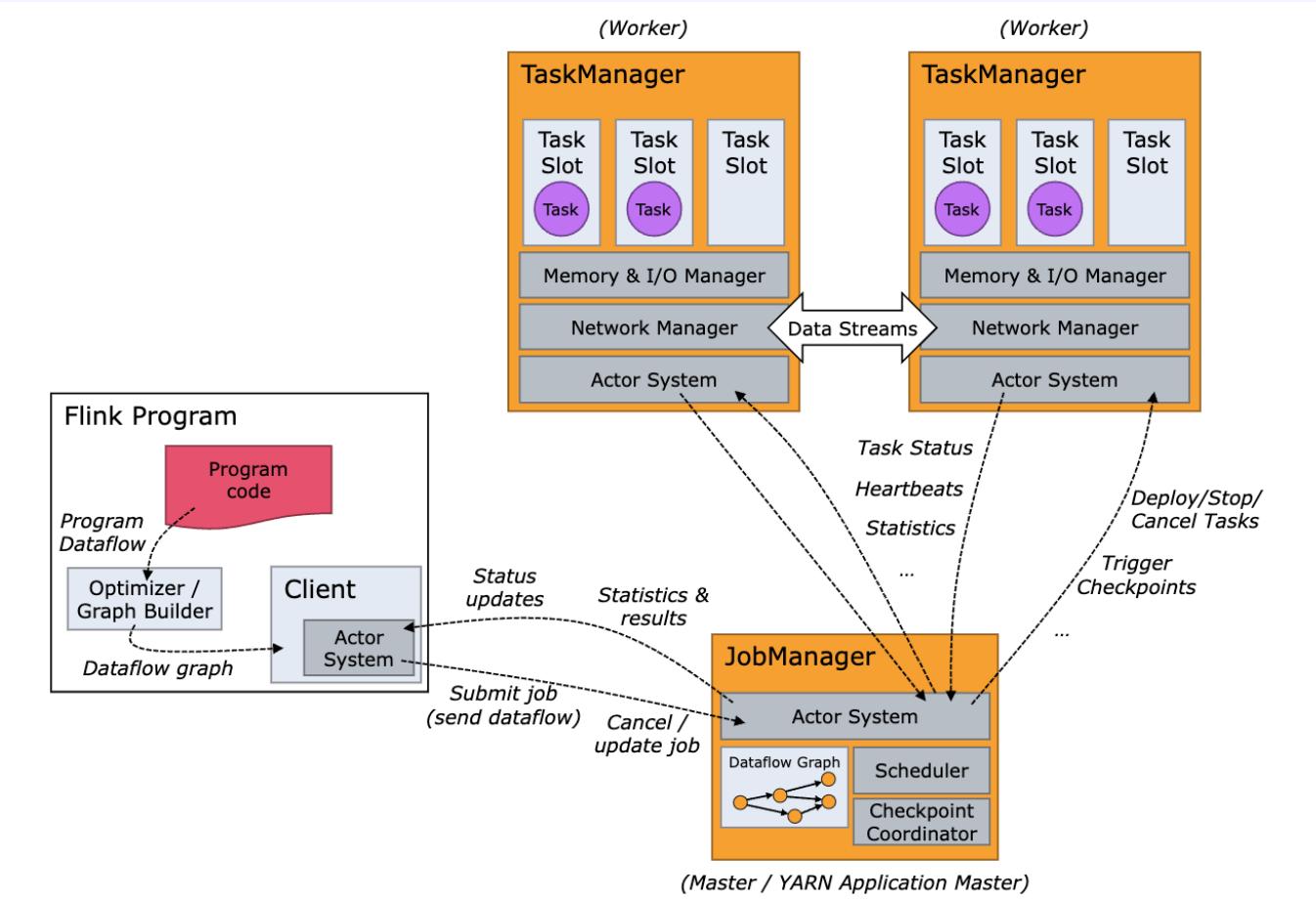
## 05. Task Slots



## 05. Task Slots



# 05. Anatomy of a Flink Cluster



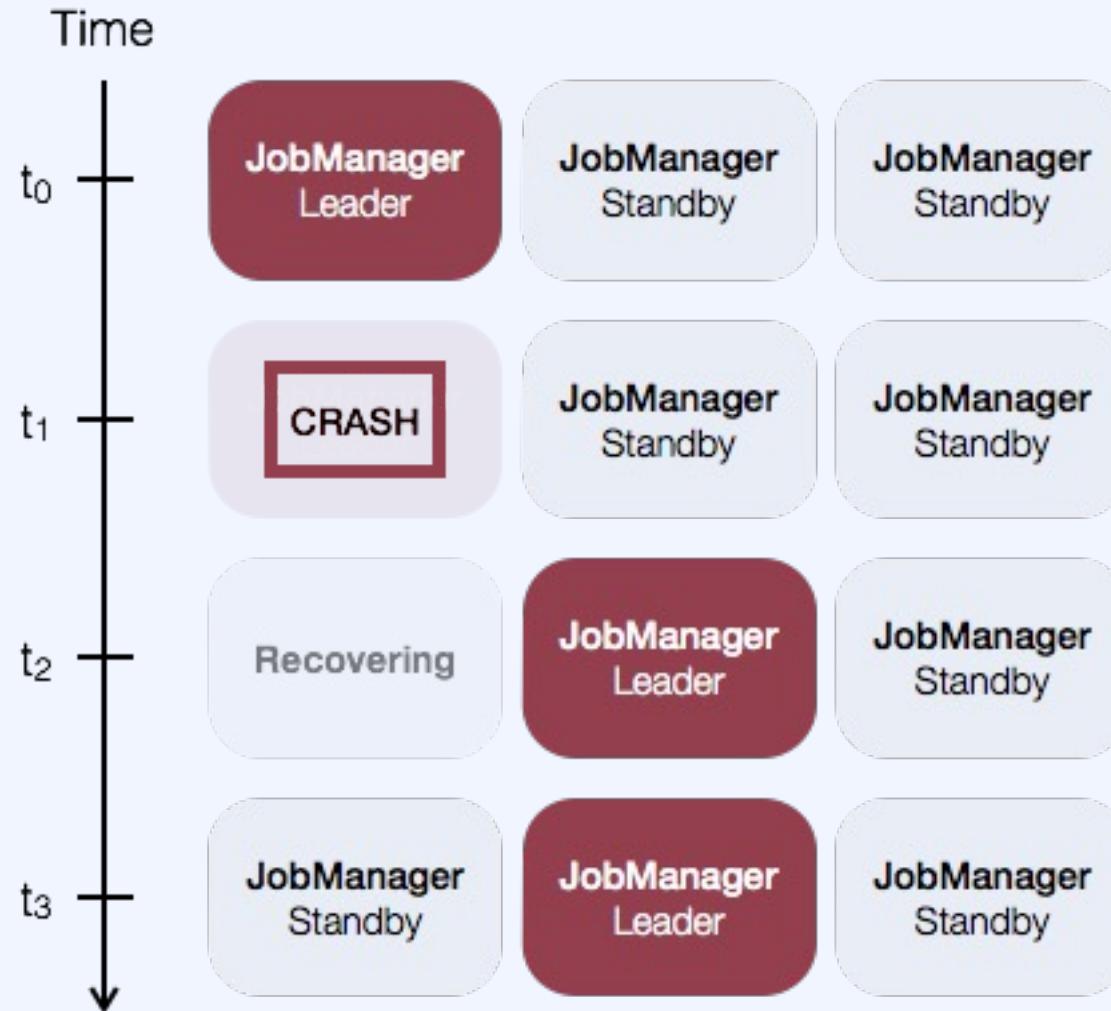
# Chapter 1.

## 06. High Availability & Execution Mode

## 06. Single Point Of Failure

- single JobManager instance per Flink cluster

## 06. Leader Selection



## 06. High Availability Services

- Zookeeper: every Flink Cluster deployment
- Kubernetes: only Kubernetes deployment

## 06. Types of Flink Cluster

- Flink Application Cluster
- Flink Job Cluster(1.15버전 이후 deprecated)
- Flink Session Cluster

## 06. 예제: Multiple Jobs in Application

```
public static void main(String arg[]){
    StreamExecutionEnvironment env1 =
        StreamExecutionEnvironment.getExecutionEnvironment();
    ...
    env1.execute(); //Job 1
    StreamExecutionEnvironment env2 =
        StreamExecutionEnvironment.getExecutionEnvironment();
    ...
    env2.execute(); //Job 2
}
```

## 06. Flink Application Cluster

- Cluster Lifecycle: dedicated cluster for **one** application
- Resource Isolation: scoped to a **single** application

## 06. Flink Session Cluster

- Cluster Lifecycle: **pre-existing**, long-running cluster
- Resource Isolation: **compete** for cluster resources
- +**a**: **Interactive** use-cases with short-running queries

# Chapter 2. DataStream API

# Chapter 2.

## 01. I/O(Source & Sink / Async)

## 01. Types of Guarantees

- At Most Once
- At Least Once
- Exactly Once

## 01. Exactly Once Guarantees

- *every event will affect the state being managed by Flink exactly once*

## 01. Exactly Once End-to-end

- Sources must be replayable
- Sinks must be transactional and idempotent

# 01. Guarantees of Sources and Sinks

Source	Guarantees	Sink	Guarantees
Apache Kafka	exactly once	Elasticsearch	at least once
AWS Kinesis Streams	exactly once	Opensearch	at least once
RabbitMQ	at most once / exactly once	Kafka producer	at least once / exactly once
Google PubSub	at least once	Cassandra sink	at least once / exactly once
Collections	exactly once	Amazon DynamoDB	at least once
Files	exactly once	Amazon Kinesis Data Streams	at least once
Sockets	at most once	File sinks	exactly once
		Socket sinks	at least once
		Standard output	at least once
		Redis sink	at least once

## 01. JDBC Sink

- At Least Once(기본)
- Exactly Once
  - upsert SQL statements
  - idempotent SQL updates

## 01. Async I/O가 필요한 이유

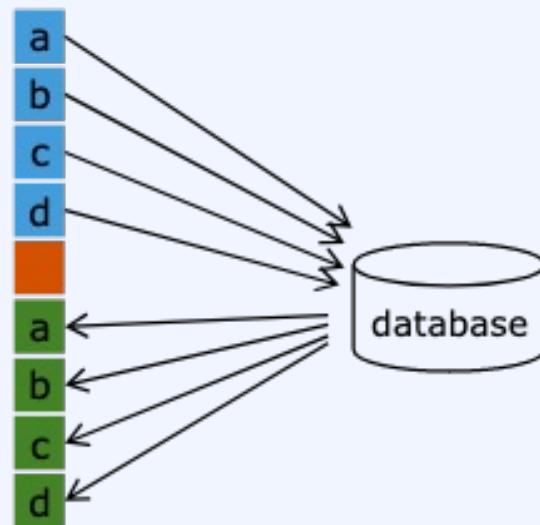
- Communication delays

# 01. Sync I/O vs Async I/O

Sync. I/O



Async. I/O



sendRequest(x)

receiveResponse(x)

wait

# 01. Async I/O API

- AsyncFunction Class 구현
- ResultFuture Callback 함수 생성
- 생성한 Async I/O Operator 적용

## 01. 예제: Implement Async I/O

```
public class AsyncDatabaseRequest extends RichAsyncFunction<String, String> {  
    transient DatabaseClient client;  
    ...  
    @Override public void asyncInvoke(String key, ResultFuture<String> future)  
    {  
        client.query(key, (DatabaseClient.Callback) (result) -> {  
            future.complete(Collections.singleton(result));  
        });  
    }  
    ...
```

## 01. 예제: Implement Async I/O

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

```
AsyncFunction<String, String> function = new AsyncDatabaseRequest();
```

```
DataStream<String> result =  
    AsyncDataStream.unorderedWait(stream, function, 1000,  
    TimeUnit.MILLISECONDS, 100);
```

## 01. AsyncDataStream의 Method

- orderedWait(input, asyncFuntion, timeout, timeUnit, capacity)
- unOrderedWait(input, asyncFuntion, timeout, timeUnit, capacity)
- orderedWaitWithRetry(input, asyncFuntion, timeout, timeUnit, capacity,  
retryStrategy, failOnOverflow)
- unOrderedWaitWithRetry(input, asyncFuntion, timeout, timeUnit, capacity,  
retryStrategy, failOnOverflow)

# 01. Async I/O Guarantees

- Full Exactly Once Fault Tolerance

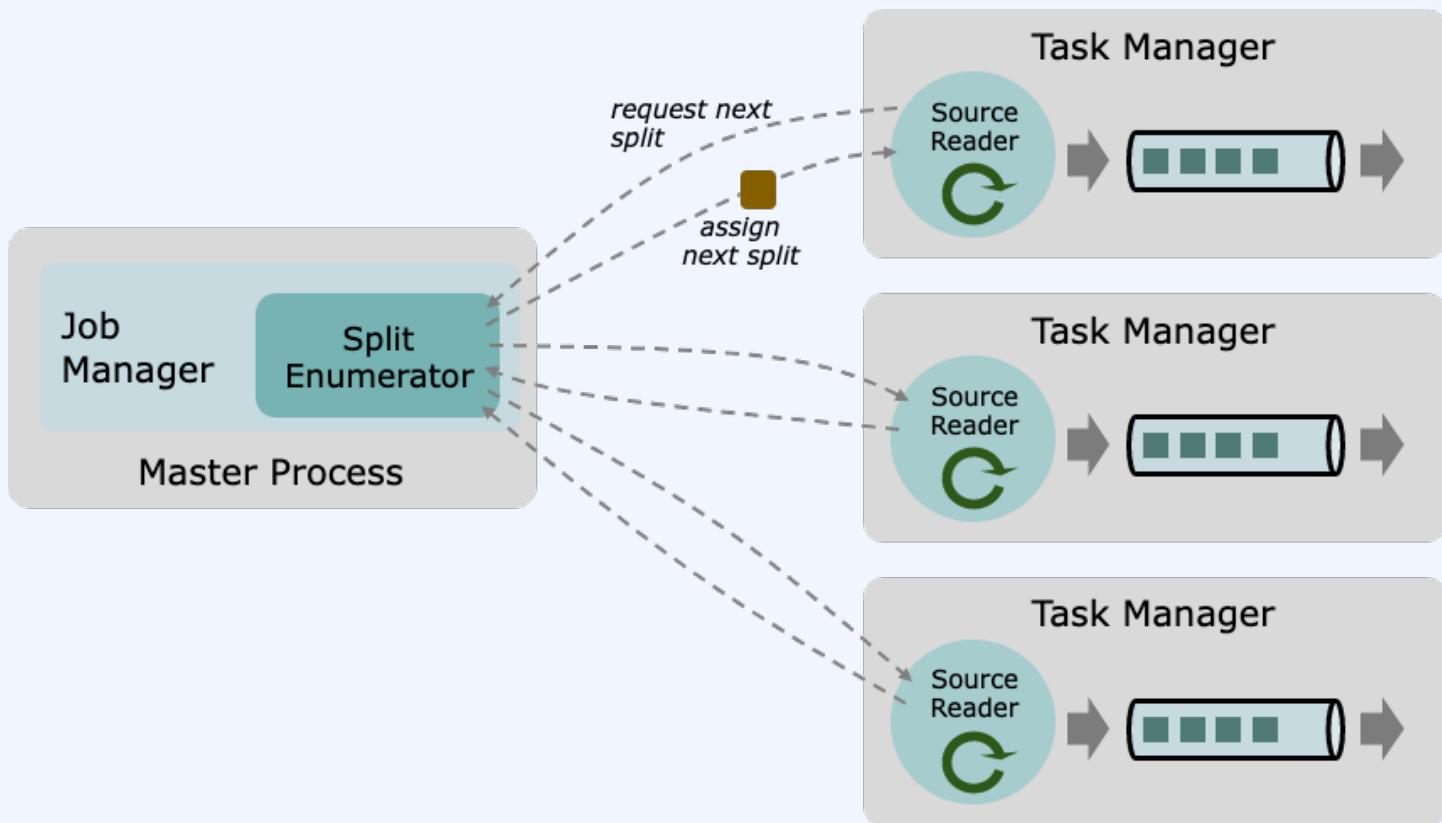
# Chapter 2.

## 02. Data Source

## 02. Core Component

- Split
- SourceReader
- SplitEnumerator

## 02. Core Component



## 02. Source Interface

```
@Public  
public interface Source<T, SplitT extends SourceSplit, EnumChkT> extends  
SourceReaderFactory<T, SplitT> {  
    Boundedness getBoundedness();  
    ...  
}
```

## 02. Unbonded Data Source Examples

- Unbounded Streaming File Source
- Unbounded Streaming Kafka Source
- 과정
  - Split Enumeration: SplitEnumerator
  - Split Assignment: SplitEnumerator & SourceReader
  - Continuous Enumeration: SplitEnumerator
  - Data Reading: SourceReader

## 02. Source Interface

```
environment.fromSource(  
    Source<OUT, ?, ?> source,  
    WatermarkStrategy<OUT> timestampsAndWatermarks,  
    String sourceName);
```

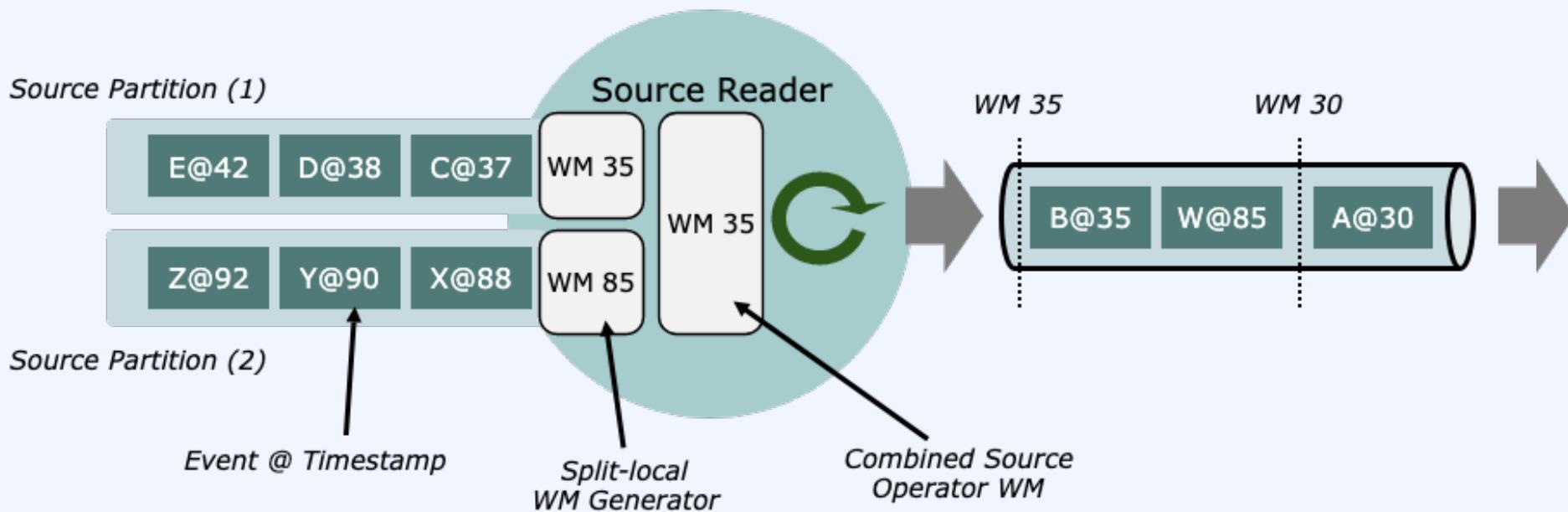
## 02. Event Timestamps

- SourceReader
  - SourceOutput.collect(event, timestamp)
  - record 기반의 timestamp가 있는 data source만
    - Kafka, Kinesis, Pulsar, Pravega
- TimestampAssigner
  - source의 record timestamp
  - event의 field에 접근

## 02. Source Interface

```
DataStream<Tuple2<String, Long>> withTimestampsAndWatermarks =  
stream  
    .assignTimestampsAndWatermarks(  
        WatermarkStrategy  
            .<Tuple2<String, Long>>forMonotonousTimestamps()  
            .withTimestampAssigner((event, timestamp) -> event.f1)  
    );
```

## 02. Watermark Generation



# Chapter 3. Operators

# Chapter 3.

## 01. Data Types & Serialization

# 01. Data Types & Serialization

- Type descriptors
- Generic type extraction
- Type serialization framework

## 01. Supported Data Types

- Java Tuples and Scala Case Classes
- Java POJOs
- Primitive Types
- Regular Classes
- Values
- Hadoop Writables
- Special Types

## 01. 예제: Java Tuples

```
DataStream<Tuple2<String, Integer>> wordCounts =  
    env.fromElements(  
        new Tuple2<String, Integer>("hello", 1),  
        new Tuple2<String, Integer>("world", 2)  
    );
```

```
wordCounts.map(value -> value.f1);
```

```
wordCounts.keyBy(value -> value.f0);
```

## 01. Java POJOs

- 클래스는 public이어야 함
- 인수가 없는 public 생성자가 있어야 함 (default constructor)
- 모든 필드는 public이거나  
getter와 setter를 통해 접근 가능해야 함
  - getFoo()과 setFoo()로 명명
- 필드의 type은 시스템에 등록된 serializer로 지원되어야 함

## 01. 예제: Java POJOs

```
public class WordWithCount {  
    public String word;  
    public int count;  
  
    public WordWithCount() {}  
    public WordWithCount(String word, int count) {  
        this.word = word; this.count = count;  
    }  
}
```

## 01. 예제: Java POJOs

```
DataStream<WordWithCount> wordCounts =  
    env.fromElements(  
        new WordWithCount("hello", 1),  
        new WordWithCount("world", 2)  
    );  
  
wordCounts.keyBy(value -> value.word);
```

## 01. Types with Built-in Serializers

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.Character`
- `java.lang.String`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.util.Date`
- `java.util.ArrayList`
- `java.util.HashMap`
- `java.util.HashSet`
- `java.util.LinkedList`
- `java.util.List`
- `java.util.Map`
- `java.util.Set`
- `org.apache.flink.api.java.tuple.Tuple1` to `org.apache.flink.api.java.tuple.Tuple25`
- `org.apache.flink.types.Row`
- `etc`

## 01. Test POJO Requirements

- flink-test-utils 라이브러리
  - org.apache.flink.types.PojoTestUtils#assertSerializedAsPojo(Class<T> clazz)
  - org.apache.flink.types.PojoTestUtils  
    #assertSerializedAsPojoWithoutKryo(Class<T> clazz)

## 01. {Pojo, Kryo, Avro}Serializer & TypeInfo

- PojoSerializer: POJO 객체를 (역)직렬화하는데 사용
  - KryoSerializer를 대체 옵션을 사용
- PojoTypeInfo: POJO의 구조와 필드의 type 정보 유지
- 예외
  - Avro types(Avro Specific Records)이거나 Avro Reflect Types인 경우
  - AvroSerializer 및 AvroTypeInfo

# 01. Type Erasure in Java

- Type Erasure in Java
- Retrieving Type Information in Flink
- Inform Type by Programmer
  - Built in method or class
    - 예를 들면,
    - StreamExecutionEnvironment.fromCollection()
    - MapFunction<I, O>
  - ResultTypeQueryable interface

## 01. 예제: Type Hints in Java API

### - Class

```
List<String> list = Arrays.asList("hello", "world");  
DataStream<String> dataStream = env.fromCollection(list, String.class);
```

### - TypeHint

```
DataStream<SomeType> result = stream  
.map(new MyGenericNonInferableFunction<Long, SomeType>())  
.returns(new TypeHint<Tuple2<Integer, SomeType>>(){});
```

# Chapter 3.

## 02. ProcessFunction

## 02. ProcessFunction

- 멤버 변수
  - serialVersionUID
- 메소드
  - processElement()
  - onTimer()
- 내부 클래스
  - Context {}
  - OnTimerContext{}

## 02. ProcessFunction{}

@PublicEvolving

public abstract class ProcessFunction<I, O> extends AbstractRichFunction

## 02. AbstractRichFunction{}

```
@Public  
public abstract class AbstractRichFunction implements RichFunction, Serializable  
{  
    ...  
  
    private transient RuntimeContext runtimeContext;  
  
    @Override  
    public void setRuntimeContext(RuntimeContext t) { ... }  
  
    @Override  
    public RuntimeContext getRuntimeContext() { ... }  
    ...
```

## 02. AbstractRichFunction{}

```
@Public  
public abstract class AbstractRichFunction implements RichFunction, Serializable  
{  
    ...  
  
    @Override  
    public void open(Configuration parameters) throws Exception {}  
  
    @Override  
    public void close() throws Exception {}  
}
```

## 02. serialVersionUID

```
private static final long serialVersionUID = 1L;
```

## 02. processElement() & onTimer()

```
public abstract void processElement(I value, Context ctx, Collector<O> out)  
throws Exception;  
  
public void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out)  
throws Exception {}
```

## 02. Context{} & OnTimerContext{}

```
public abstract class Context {  
    public abstract Long timestamp();  
    public abstract TimerService timerService();  
    public abstract <X> void output(OutputTag<X> outputTag, X value);  
}  
  
public abstract class OnTimerContext extends Context {  
    public abstract TimeDomain timeDomain();  
}
```

## 02. TimerService{}

- long currentProcessingTime()
- long currentWatermark()
- void deleteEventTimeTimer(long time)
- void deleteProcessingTimeTimer(long time)
- void registerEventTimeTimer(long time)
- void registerProcessingTimeTimer(long time)

## 02. processElement() & onTimer()

```
public abstract void processElement(I value, Context ctx, Collector<O> out)  
throws Exception;  
  
public void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out)  
throws Exception {}
```

## 02. Basic Building Blocks

- events: stream elements
- state: fault-tolerant, consistent
  - only on keyed stream
- timers: event time and processing time
  - only on keyed stream

## 02. processElement() & onTimer()

```
public abstract void processElement(I value, Context ctx, Collector<O> out)  
throws Exception;  
  
public void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out)  
throws Exception {}
```

## 02. Collector{}

```
/** * Collects a record and forwards it. The collector is the "push" counterpart  
of the {@link * java.util.Iterator}, which "pulls" data in. */  
@Public  
public interface Collector<T> {  
    /** * Emits a record. ** @param record The record to collect. */  
    void collect(T record);  
    /** Closes the collector. If any data was buffered, that data will be flushed. */  
    void close();  
}
```

## 02. processElement() & onTimer()

```
public abstract void processElement(I value, Context ctx, Collector<O> out)  
throws Exception;  
  
public void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out)  
throws Exception {}
```

## 02. Context{} & OnTimerContext{}

```
public abstract class Context {  
    public abstract Long timestamp();  
    public abstract TimerService timerService();  
    public abstract <X> void output(OutputTag<X> outputTag, X value);  
}  
  
public abstract class OnTimerContext extends Context {  
    public abstract TimeDomain timeDomain();  
}
```

## 02. Context{} & OnTimerContext{}

```
public abstract class Context {  
    public abstract Long timestamp();  
    public abstract TimerService timerService();  
    public abstract <X> void output(OutputTag<X> outputTag, X value);  
}  
  
public abstract class OnTimerContext extends Context {  
    public abstract TimeDomain timeDomain();  
}
```

## 02. Context{} & OnTimerContext{}

```
public abstract class Context {  
    public abstract Long timestamp();  
    public abstract TimerService timerService();  
    public abstract <X> void output(OutputTag<X> outputTag, X value);  
}  
  
public abstract class OnTimerContext extends Context {  
    public abstract TimeDomain timeDomain();  
}
```

## 02. ProcessFunction Family

- ProcessFunction
- KeyedProcessFunction
- CoProcessFunction
- KeyedCoProcessFunction
- ProcessWindowFunction
- ProcessAllWindowFunction

## 02. CoProcessFunction{}

```
@PublicEvolving
public abstract class CoProcessFunction<IN1, IN2, OUT> extends
AbstractRichFunction {
    ...
    public abstract void processElement1(IN1 value, Context ctx, Collector<OUT>
out) throws Exception;
    public abstract void processElement2(IN2 value, Context ctx, Collector<OUT>
out) throws Exception;
    ...
}
```

# Chapter 3.

## 03. Map Operator

## 03. Map

- Type
  - DataStream -> DataStream
- Takes one element and produces one element

## 03. 예제: Map

```
DataStream<Integer> dataStream = //...
dataStream.map(new MapFunction<Integer, Integer>() {
    @Override public Integer map(Integer value) throws Exception {
        return 2 * value;
    }
});
```

## 03. MapFunction Interface

```
@Public  
@FunctionalInterface  
public interface MapFunction<T, O> extends Function, Serializable {  
    O map(T value) throws Exception;  
}
```

## 03. 예제: Map (Lambda Expression)

```
DataStream<Integer> dataStream = //...  
dataStream.map(value -> 2 * value);
```

## 03. MapFunction Interface

```
@Public  
@FunctionalInterface  
public interface MapFunction<T, O> extends Function, Serializable {  
    O map(T value) throws Exception;  
}
```

## 03. Function Interface

```
@Public  
public interface Function extends java.io.Serializable {}
```

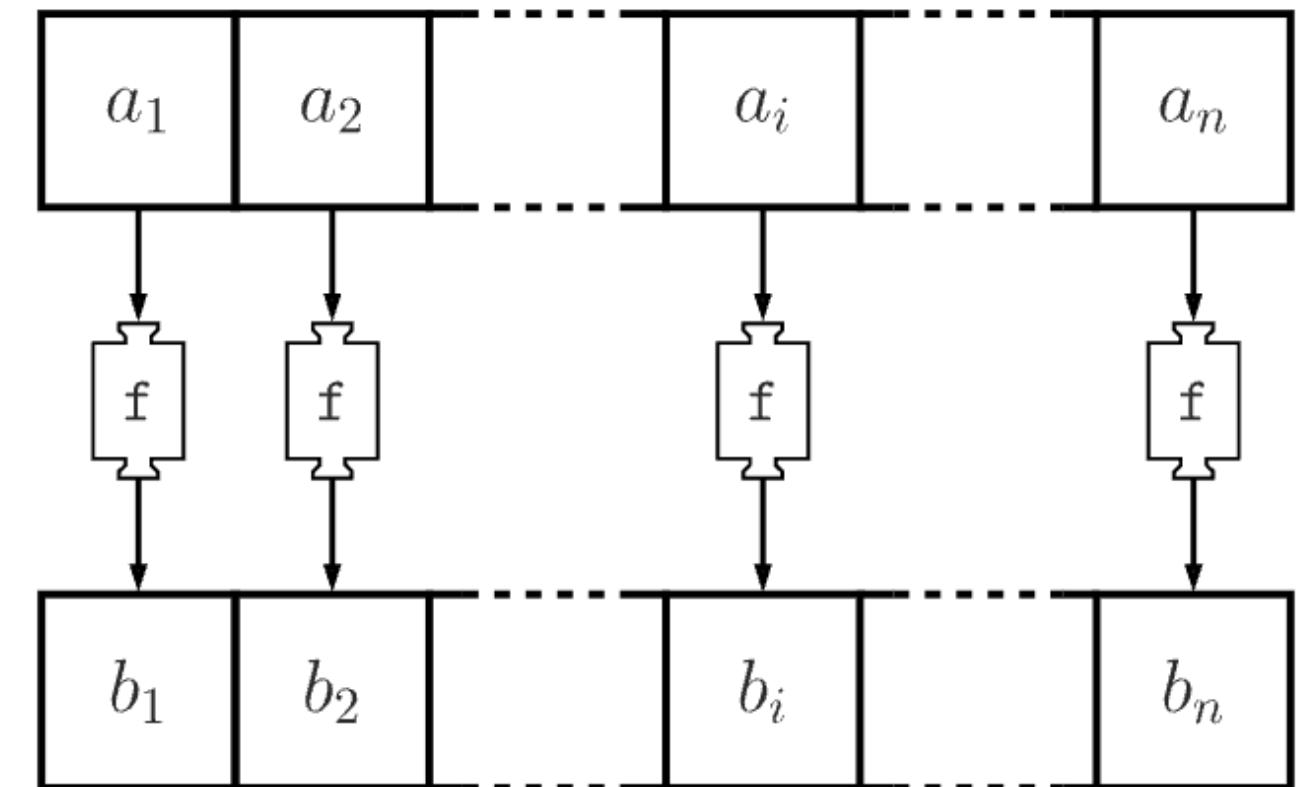
## 03. MapFunction Interface

```
@Public  
@FunctionalInterface  
public interface MapFunction<T, O> extends Function, Serializable {  
    O map(T value) throws Exception;  
}
```

## 03. MapFunction Interface

```
@Public  
@FunctionalInterface  
public interface MapFunction<T, O> extends Function, Serializable {  
    O map(T value) throws Exception;  
}
```

## 03. Map in Graphic (Scala)



## 03. Example

- 주식 중개업체는 실시간 주식 가격 업데이트를 받습니다.  
처리하는 각 거래마다 중개 수수료를 부과합니다.  
주식을 구입하는 실제 비용을 반영하기 위해, 주식 가격에 중개 수수료를 더하여 수입되는 주식 가격을 조정하려고 합니다.

## 03. Practice

- 금융 데이터 제공업체는 실시간 주식 가격을 스트리밍합니다.  
그들의 고객은 전 세계에 걸쳐 있으므로, 수신되는 주식 가격을 고객의 지역 통화로 조정하고자 합니다.  
이번 과제에서는, 고객이 유럽에 위치하고 있으며, 주식 가격을 미국 달러에서 유로로 변환하고자 합니다.
  - 상수 환율(0.85)을 사용하여 수신되는 주식 가격을 미국 달러에서 유로로 변환합니다.

# Chapter 3.

## 04. Filter Operator

## 04. Filter

- Type
  - $\text{DataStream} \rightarrow \text{DataStream}$
- Evaluates a boolean function for each element  
and retains those for which the function returns true

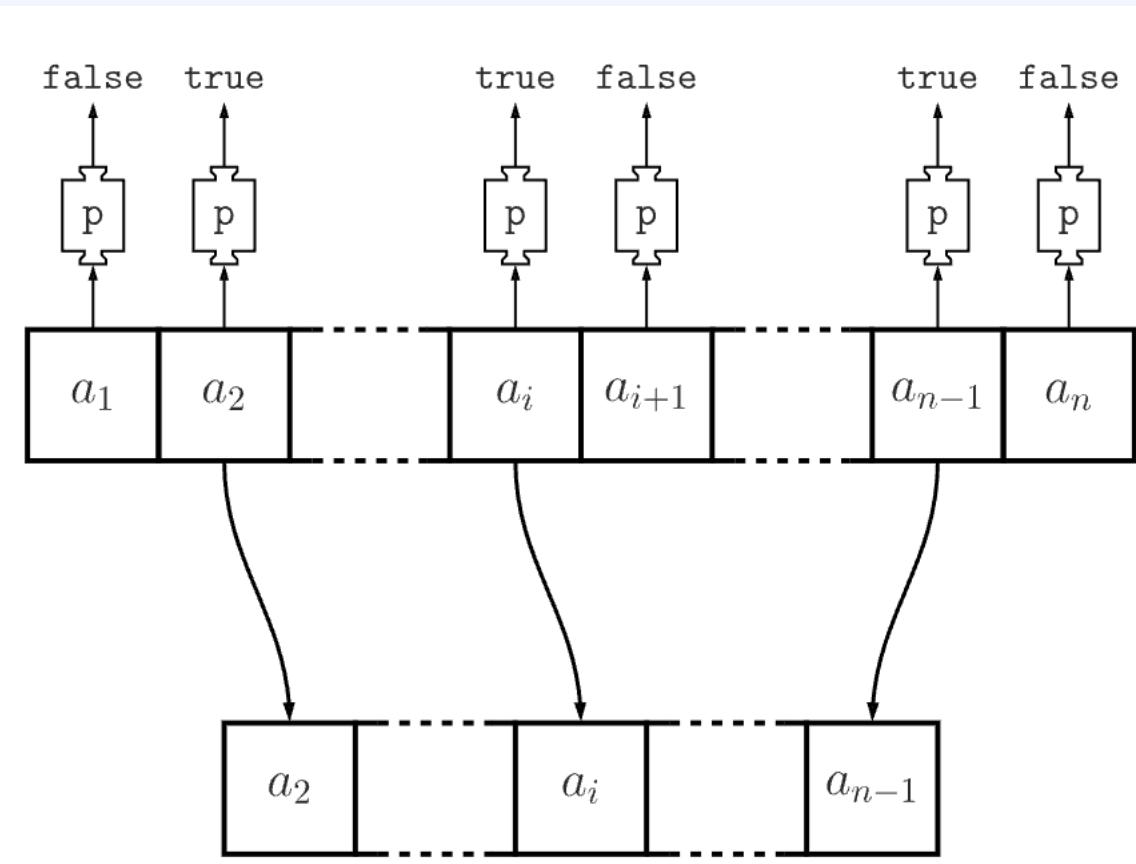
## 04. 예제: Filter

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

## 04. FilterFunction Interface

```
@Public  
@FunctionalInterface  
public interface FilterFunction<T> extends Function, Serializable {  
    boolean filter(T value) throws Exception;  
}
```

## 04. Filter in Graphic (Scala)



## 04. Example

- 금융 분야에서는 고가 주식의 분석이 저가 주식의 분석보다 더 흥미로운 경우가 많습니다. 이는 고가 주식이 시장에 더 큰 영향을 미치기 때문입니다. 예를 들어 수십만 달러에서 거래되는 Berkshire Hathaway와 같은 주식은 작은 백분율 변화라도 지수에 큰 영향을 미칠 수 있습니다. 따라서 일반적으로 일정 임계값 이상으로 거래되는 주식을 필터링하여 사용하는 경우가 많습니다.

## 04. Practice

- 당신은 주식 거래 회사에서 데이터 분석가로 일하고 있으며, 특정 주식을 모니터링하는 것에 관심이 있습니다.

회사는 Facebook(FB), Amazon(AMZN), Apple(AAPL), Netflix(NFLX) 및 Google(GOOG)과 같이 FAANG으로 자주 언급되는 특정 회사의 주식을 신중하게 모니터링하고자 합니다. 이 목록에 없는 주식을 필터링하는 것이 업무로 맡겨졌습니다.

# Chapter 3.

## 05. FlatMap Operator

## 05. FlatMap

- Type
  - DataStream -> DataStream
- Takes one element and produces zero, one, or more elements

## 05. 예제: FlatMap

```
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);  
        }  
    }  
});
```

## 05. FlatMapFunction Interface

```
@Public  
@FunctionalInterface  
public interface FlatMapFunction<T, O> extends Function, Serializable {  
    void flatMap(T value, Collector<O> out) throws Exception;  
}
```

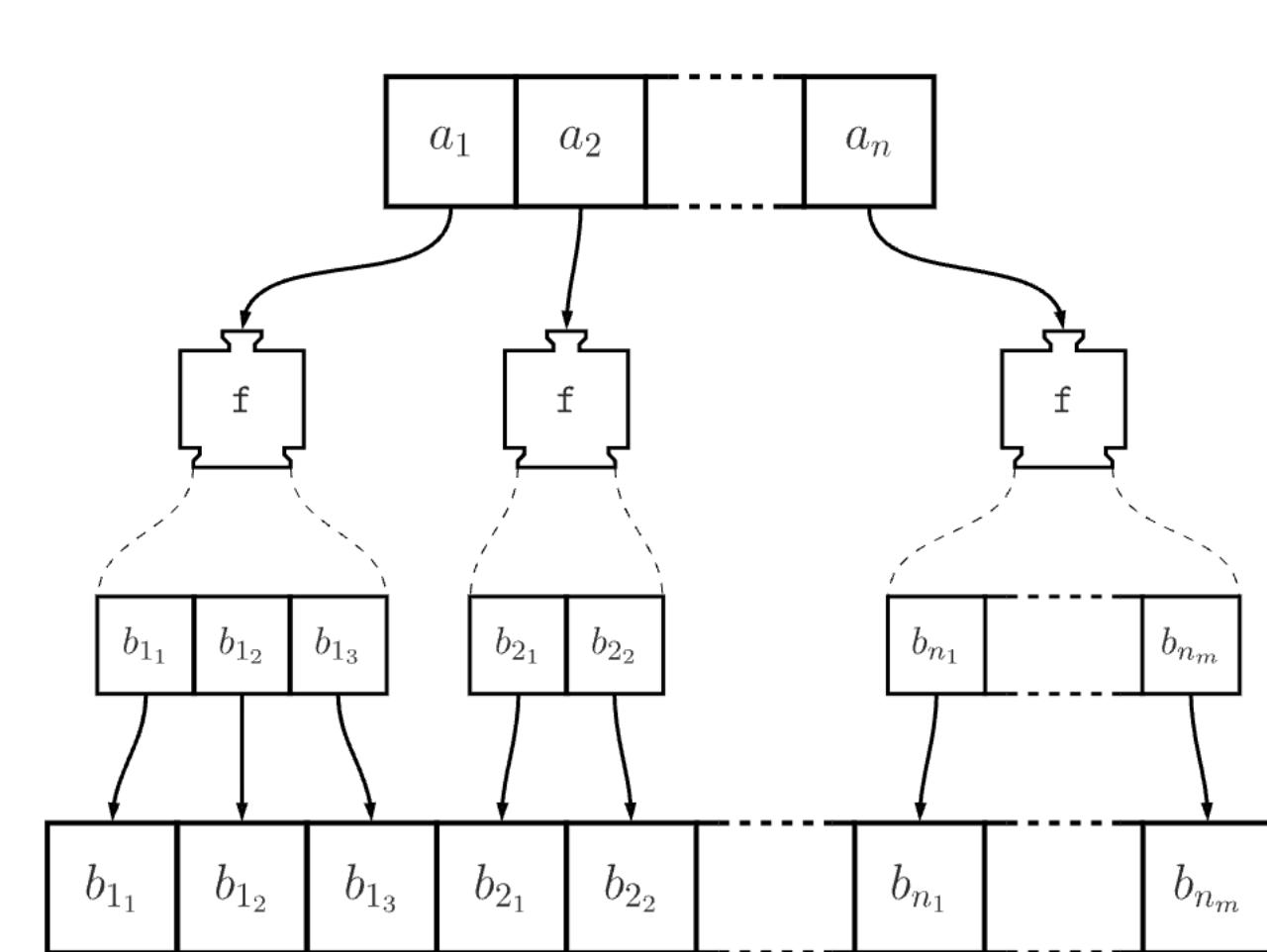
## 05. 예제: FlatMap

```
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);  
        }  
    }  
});
```

## 05. FlatMapFunction Interface

```
@Public  
@FunctionalInterface  
public interface FlatMapFunction<T, O> extends Function, Serializable {  
    void flatMap(T value, Collector<O> out) throws Exception;  
}
```

## 05. FlatMap in Graphic (Scala)



## 05. Example

- 금융 산업에서는 효율성을 위해 주식 거래소 데이터 피드에서 한 번의 메시지로 여러 주식에 대한 업데이트를 제공하는 것이 일반적입니다. 특히 최대 거래 시간 동안에는 이런 형태로 각 레코드가 여러 주식에 대한 정보를 콤마로 구분된 문자열 형태로 포함하고 있을 것입니다.  
이 경우, 이러한 다중 주식 레코드를 각 주식별로 개별 레코드로 분할해야 합니다.

## 05. Practice

- 각 주식의 섹터 정보, 즉 해당 주식이 속한 섹터를 나타내는 정보가 포함된 스트리밍 주식 데이터가 있는 시나리오를 고려해 봅시다.  
즉, 주식 심볼, 가격 및 타임스탬프에 추가로 해당 주식이 속한 섹터 정보가 포함된 것입니다.  
때로는 같은 섹터에 속한 여러 주식이 한 번에 하나의 메시지로 업데이트됩니다. 정보는 다음  
과 같은 형식으로 제공됩니다.  
``sector:symbol1:price1:timestamp1,symbol2:price2:timestamp2,...``  
하나의 메시지를 여러 섹터 주식으로 분할해야 합니다.

# Chapter 3.

## 06. KeyBy Operator

## 06. KeyBy

- Type
  - DataStream -> KeyedStream
- Logically partitions a stream into disjoint partitions.  
(Internally, keyBy() is implemented with hash partitioning.)

## 06. 예제:KeyBy

```
dataStream.keyBy(value -> value.getSomeKey());  
dataStream.keyBy(value -> value.f0);
```

## 06. DataStream#keyBy()

```
@Public  
public class DataStream<T> {  
    ...  
    public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key) {  
        Preconditions.checkNotNull(key);  
        return new KeyedStream(this, (KeySelector)this.clean(key));  
    }  
    ...
```

## 06. DataStream#keyBy()

```
@Public  
public class DataStream<T> {  
    ...  
    public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key) {  
        Preconditions.checkNotNull(key);  
        return new KeyedStream(this, (KeySelector)this.clean(key));  
    }  
    ...
```

## 06. KeySelector Interface

```
@FunctionalInterface  
@Public  
public interface KeySelector<IN, KEY> extends Function, Serializable {  
    KEY getKey(IN var1) throws Exception;  
}
```

## 06. DataStream#keyBy()

```
@Public  
public class DataStream<T> {  
    ...  
    public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key) {  
        Preconditions.checkNotNull(key);  
        return new KeyedStream(this, (KeySelector)this.clean(key));  
    }  
    ...
```

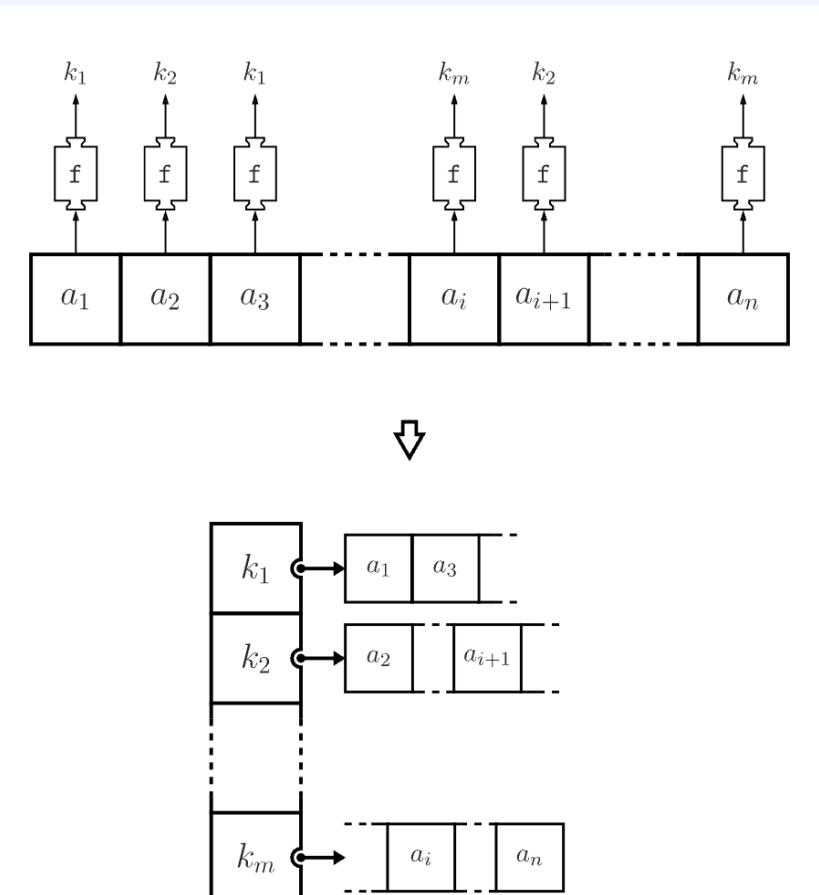
## 06. DataStream#keyBy()

```
@Public  
public class DataStream<T> {  
    ...  
    public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key) {  
        Preconditions.checkNotNull(key);  
        return new KeyedStream(this, (KeySelector)this.clean(key));  
    }  
    ...
```

## 06. DataStream#keyBy()

```
@Public  
public class DataStream<T> {  
    ...  
    public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key) {  
        Preconditions.checkNotNull(key);  
        return new KeyedStream(this, (KeySelector)this.clean(key));  
    }  
    ...
```

## 06. KeyBy in Graphic (Scala)



## 06. Example

- 여러분은 주식 중개회사에서 데이터 분석가로 일하고 있습니다.  
이 중개회사는 실시간 주식 데이터 스트림을 받으며, 여러 주식 심볼별로 이 데이터를 분석하는 것이 여러분의 업무입니다.  
수신되는 데이터에는 다양한 주식 심볼이 포함되어 있으며, 각 주식에 대해 데이터를 구분하여 개별 주식에 대한 추가적인 분석을 수행하려고 합니다.

## 06. Practice

- 여러분의 업무는 실시간으로 각 주식 심볼에 대한 거래 횟수를 추적하는 것입니다.

# Chapter 3.

## 07. Reduce Operator

## 07. Reduce

- Type
  - KeyedStream -> DataStream
- A “rolling” reduce
  - Combines the current element with the last reduced value and emits the new value.

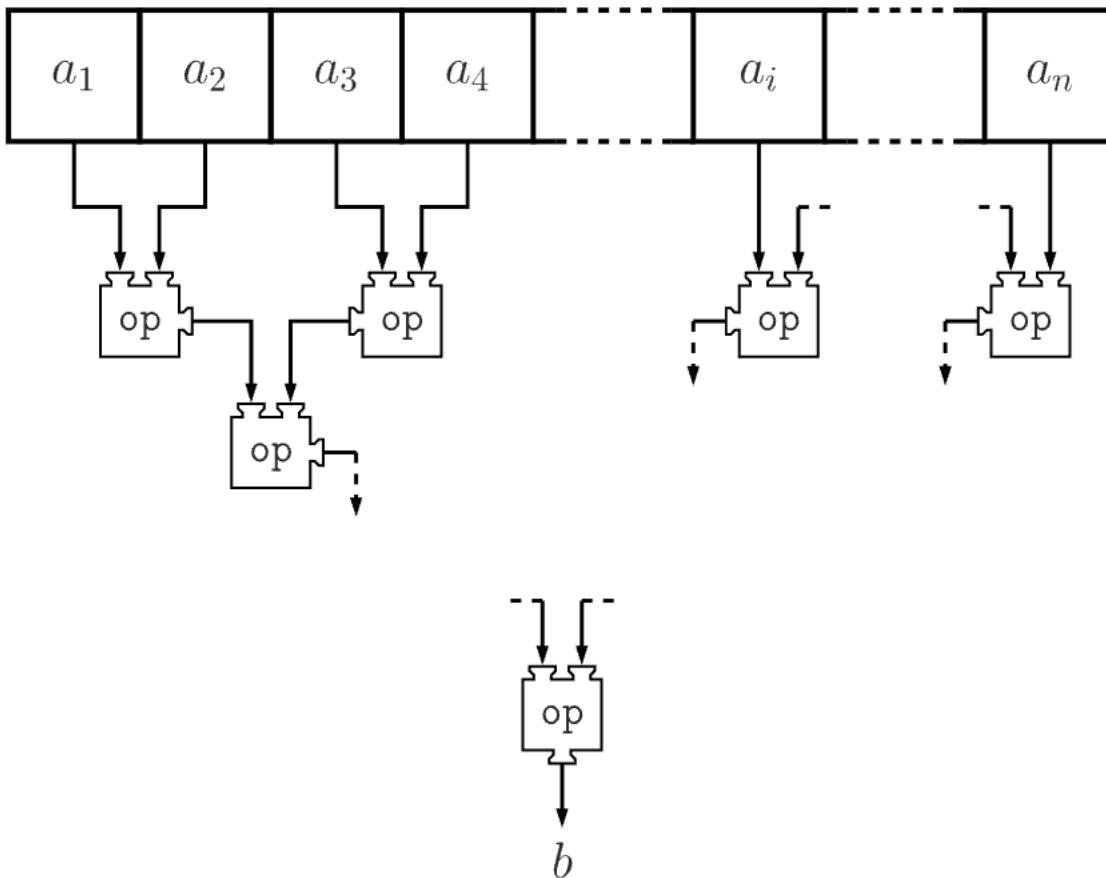
## 07. 예제: Reduce

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

## 07. ReduceFunction Interface

```
@Public  
@FunctionalInterface  
public interface ReduceFunction<T> extends Function, Serializable {  
    T reduce(T value1, T value2) throws Exception;  
}
```

## 07. Reduce in Graphic (Scala)



## 07. Example

- 주식 거래 회사에서, 한 분석가가 하루 동안 각 주식 심볼에 대한 총 거래량에 관심이 있습니다. 이 분석가는 실시간 데이터에만 관심이 있으며, 거래가 하루 동안 발생할 때마다 주식별 누적 거래량을 얻고자 합니다.

## 07. Practice

- 증권 회사에서, 위험 관리 부서는 각 주식의 실시간 최대 거래량을 모니터링하고자 합니다. 이는 잠재적인 시장 조작이나 내부자 거래를 나타낼 수 있는 비정상적인 거래 활동을 신속하게 감지하기 위함입니다.

# Chapter 3.

## 08. Windows

## 08. Windows

- 어떤 함수들은 한정된 데이터에 대해서만 의미를 가짐
  - 정렬
  - 집계: sum, average, min, max, ⋯
  - 집합: union, intersection, difference, ⋯
  - ML
  - Join: full outer join, ⋯

## 08. Types of Windows

- Keyed streams: “groups” stream elements to process in parallel
- Non-keyed streams: non-parallel(parallelism 1) processing of original stream

## 08. 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/apply() <- required: "function"  
[.getSideOutput(...)] <- optional: "output tag"
```

## 08. 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/apply() <- required: "function"

[.getSideOutput(...)] <- optional: "output tag"

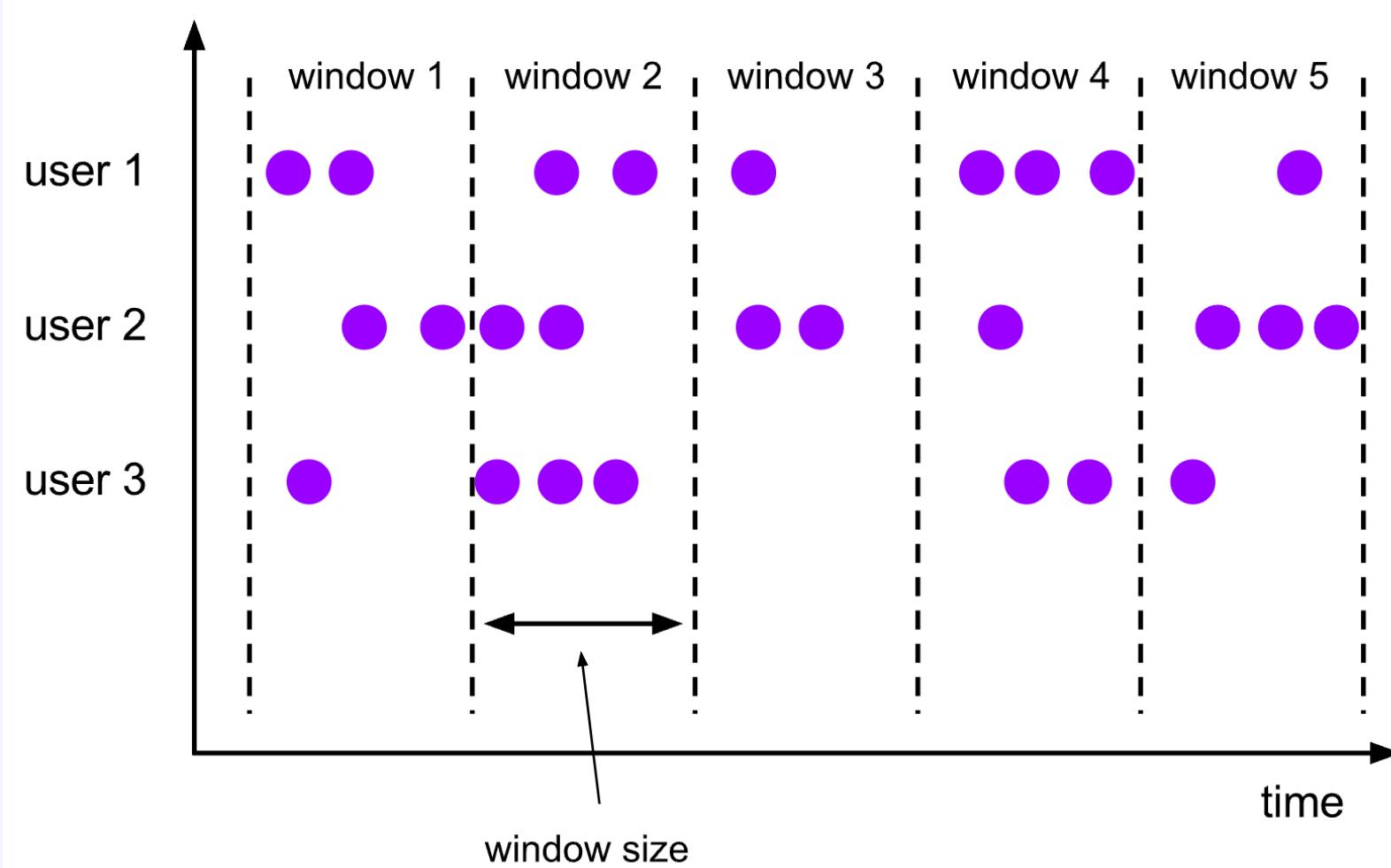
## 08. Lifecycle of Windows

- Window Creation
- Window Population
- Window Triggering: CountWindow, TimeWindow
- Pre-Function Eviction(optional)
- Function Application
- Post-Function Eviction(optional)
- Window Removal: End of Window + Allowed Lateness

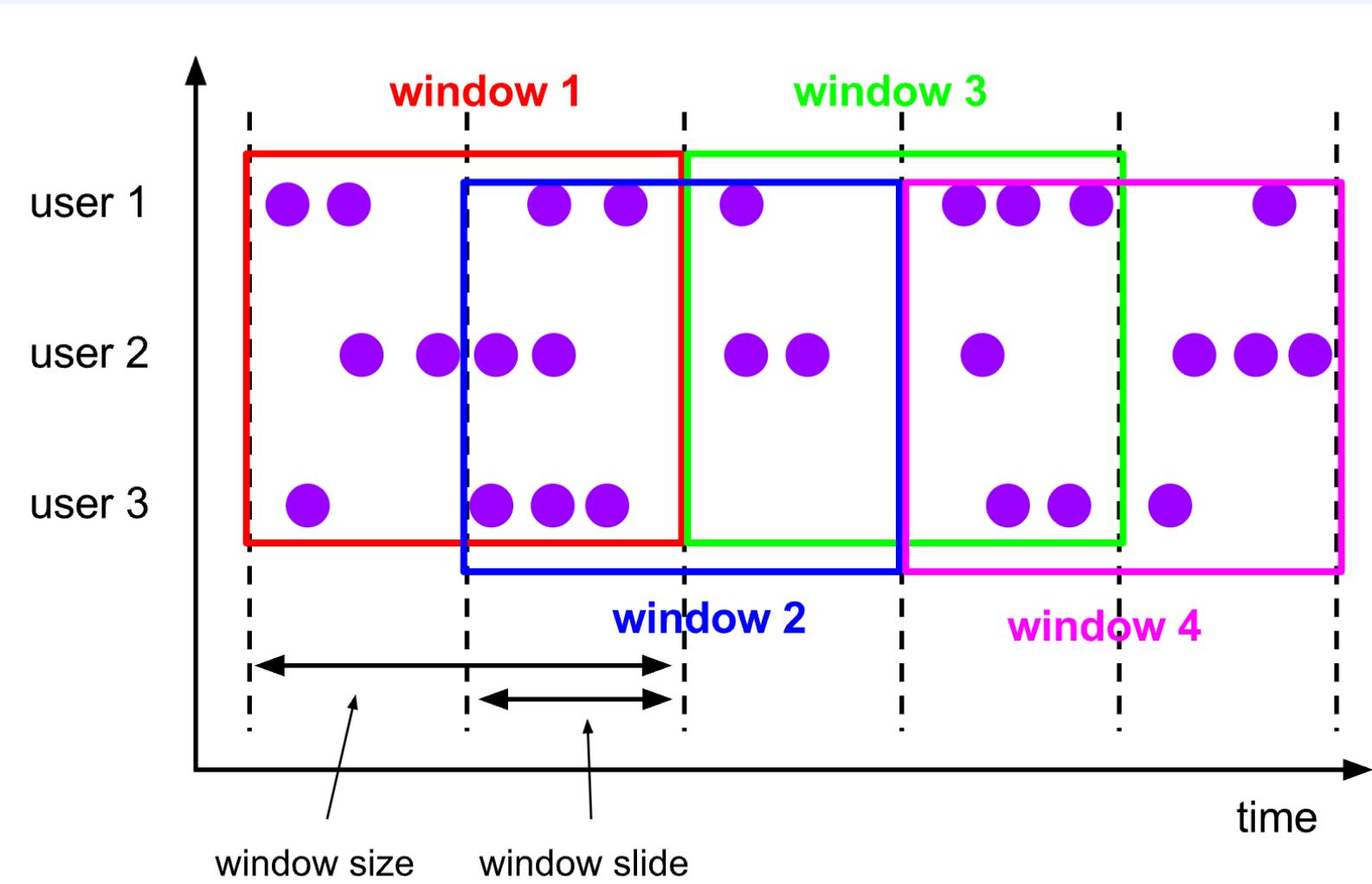
## 08. Window Assigners

- Tumbling Windows
- Sliding Windows
- Session Windows
- Global Windows

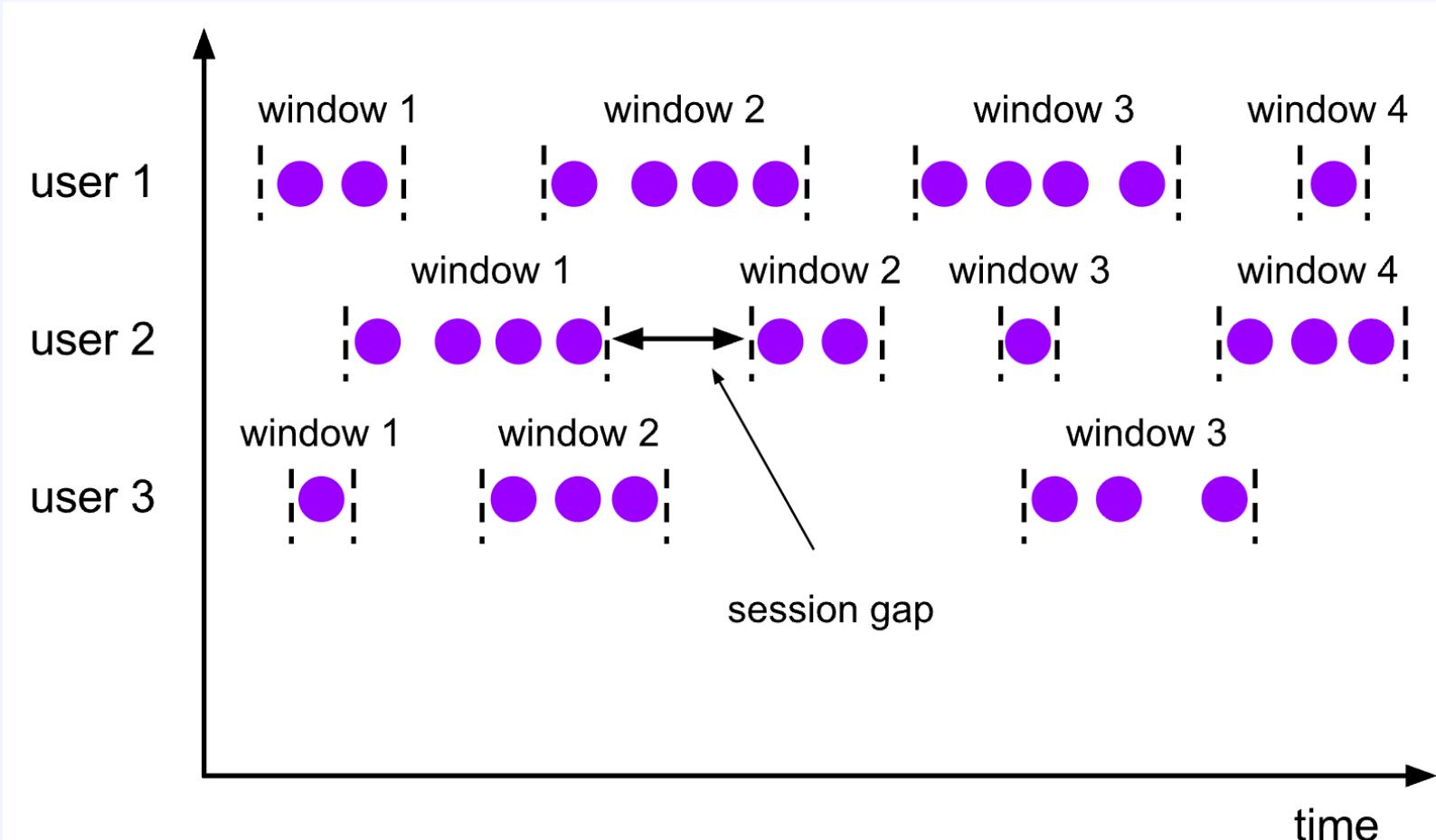
## 08. Tumbling Windows



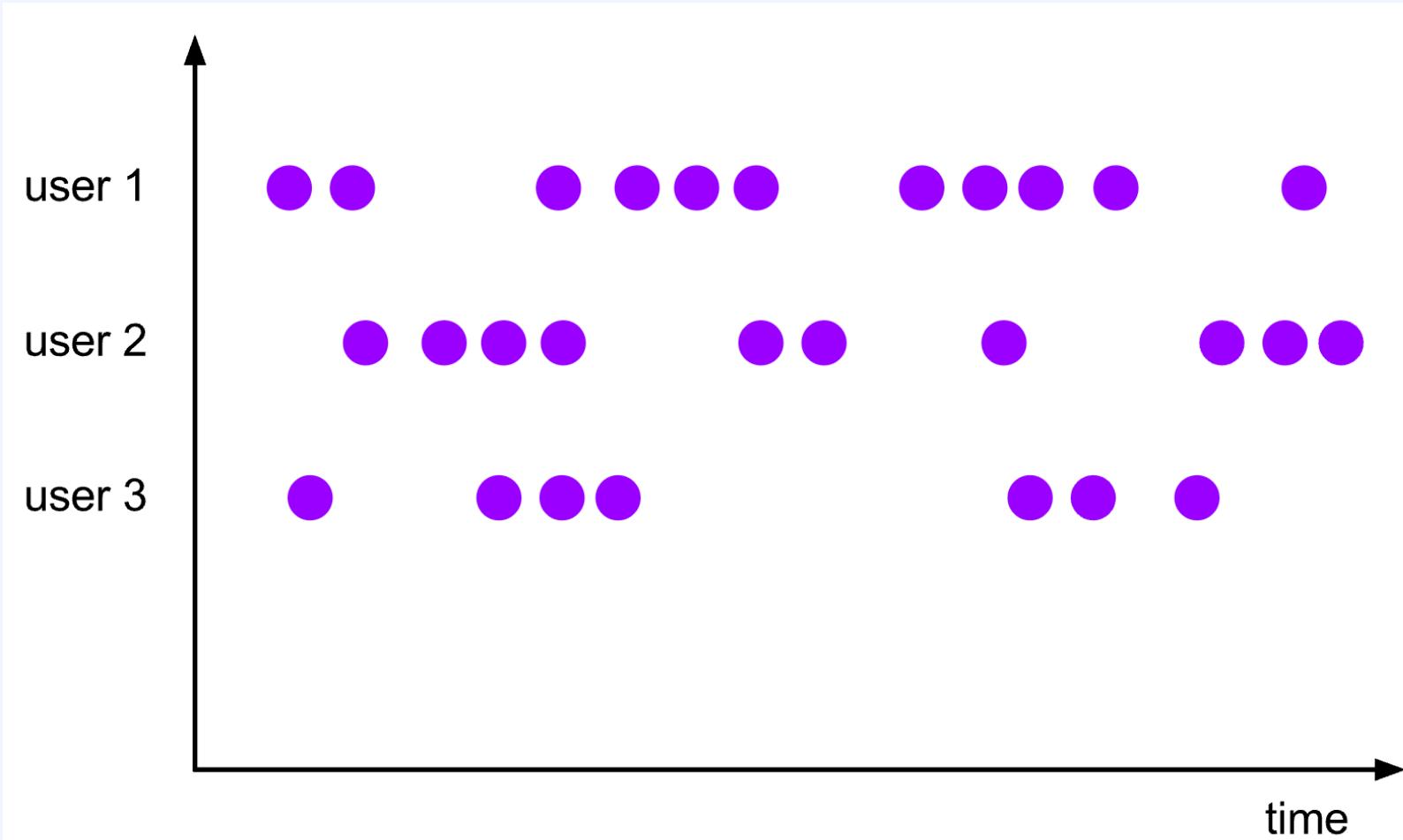
## 08. Sliding Windows



## 08. Session Windows



## 08. Global Windows



## 08. Window Assigners

**Tumbling time windows**



**Sliding time windows**



**Tumbling count windows**



**Sliding count windows**



**Session windows**



**Global window**



## 08. Built-in Triggers

- EventTimeTrigger
- ProcessingTimeTrigger
- ContinuousEventTimeTrigger
- ContinuousProcessingTimeTrigger
- CountTrigger
- DeltaTrigger
- PurgingTrigger(WrappingTrigger)

## 08. 예제: PurgingTrigger, CountTrigger

stream

```
.keyBy((KeySelector<MyEvent, String>) MyEvent::getKey)
.window(GlobalWindows.create())
.trigger(PurgingTrigger.of(CountTrigger.of(5)))
.process(new MyProcessWindowFunction());
```

## 08. Trigger Interface

- Methods
  - `onElement()`
  - `onEventTime()`
  - `onProcessingTime()`
  - `onMerge()`
  - `clear()`

## 08. Trigger Interface

- Methods에 대해 알아야 할 두 가지 사항
  - 작업 방식을 결정하기 위해 TriggerResult 반환  
(onElement(), onEventTime(), onProcessingTime())
    - CONTINUE
    - FIRE
    - PURGE
    - FIRE\_AND\_PURGE
  - processing- 및 event-time timers 등록 가능  
(모든 메소드)

## 08. Window Functions

Operation	Description
reduce	aggregate the elements of a window (input type == output type)
aggregation (sum/min/max)	generalized version of a reduce operation (get sum/min/max of elements in a window)
process	Generic function to process iterable(elements) in a window

## 08. Evictor Interface

- Methods
  - `void evictBefore(Iterable<TimestampedValue<T>> elements, int size, W window, EvictorContext evictorContext);`
  - `void evictAfter(Iterable<TimestampedValue<T>> elements, int size, W window, EvictorContext evictorContext);`

## 08. Pre-implemented evictors

- Before the window function
  - CountEvictor: last 'N' elements
  - DeltaEvictor: within certain delta
  - TimeEvictor: last 'N' milliseconds

## 08. 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/apply() <- required: "function"  
[.getSideOutput(...)] <- optional: "output tag"
```

## 08. Interaction of watermarks and windows

- When watermarks arrive at the window operator
  - the watermark **triggers** computation of all windows where the maximum timestamp (which is end-timestamp - 1) is smaller than the new watermark
  - the watermark is forwarded (as is) to downstream operations

## 08. 예제: Consecutive Windowed Operations

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

```
DataStream<Integer> resultsPerKey = input  
    .keyBy(<key selector>)  
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))  
    .reduce(new Summer());
```

```
DataStream<Integer> globalResults = resultsPerKey  
    .windowAll(TumblingEventTimeWindows.of(Time.seconds(5)))  
    .process(new TopKWindowFunction());
```

## 08. State Size Considerations

- Element Replication
- Use of Aggregation Functions
- Effect of Evictors

# Chapter 3.

## 09. ProcessWindowFunction

## 09. ProcessWindowFunction

- 멤버 변수
  - serialVersionUID
- 메소드
  - process()
  - clear()
- 내부 클래스
  - Context {}

## 09. ProcessFunction과 비교

- process() vs processElement()
- process() vs onTimer()

## 09. ProcessWindowFunction{}

```
@PublicEvolving
public abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window>
extends AbstractRichFunction {
```

## 09. Window{}

```
@PublicEvolving  
public abstract class Window {  
    public abstract long maxTimestamp();  
}
```

## 09. TimeWindow{}

```
@PublicEvolving
public class TimeWindow extends Window {
    ...
    public TimeWindow(long start, long end) { this.start = start; this.end = end; }
    @Override
    public long maxTimestamp() { return end - 1; }
    @Override
    public boolean equals(Object o) { ... }
    @Override
    public int hashCode() { ... }
    ...
}
```

## 09. process()

```
public abstract void process(KEY key, Context context, Iterable<IN> elements,  
Collector<OUT> out) throws Exception;
```

- Late Firings
  - End of Window < process() <= End of Window + Allowed Lateness

## 09. clear()

```
public void clear(Context context) throws Exception {
```

- Window expires
  - Watermark가 maxTimestamp + allowedLateness를 지날 때

## 09. Context{}

```
public abstract class Context implements java.io.Serializable {  
    public abstract W window();  
    public abstract long currentProcessingTime();  
    public abstract long currentWatermark();  
    public abstract KeyedStateStore windowState();  
    public abstract KeyedStateStore globalState();  
    public abstract <X> void output(OutputTag<X> outputTag, X value);  
}
```

- Key1: Key1-Window1 + Key1-Window2 + ⋯ + Key1-WindowN

## 09. clear()

```
public void clear(Context context) throws Exception {}
```

## 09. process()

```
public abstract void process(KEY key, Context context, Iterable<IN> elements,  
Collector<OUT> out) throws Exception;
```

## 09. ProcessAllWindowFunction{}

```
@PublicEvolving
public abstract class ProcessAllWindowFunction<IN, OUT, KEY, W extends
Window> extends AbstractRichFunction {
    ...
    public abstract void process(KEY key, Context context, Iterable<IN> elements,
        Collector<OUT> out) throws Exception;
    ...
    public abstract long currentProcessingTime();
    public abstract long currentWatermark();
    ...
}
```

# Chapter 3.

## 10. Window Operator

## 10. Window

- Type
  - KeyedStream -> WindowedStream
- Groups the data in each key according to some characteristic  
e.g., the data that arrived within the last 5 seconds

## 10. 예제: Window

dataStream

```
.keyBy(value -> value.f0)
```

```
.window(TumblingEventTimeWindows.of(Time.seconds(5)));
```

## 10. KeyedStream#window()

```
@Public  
public class KeyedStream<T, KEY> extends DataStream<T> {  
    ...  
  
    @PublicEvolving  
    public <W extends Window> WindowedStream<T, KEY, W>  
        window(WindowAssigner<? super T, W> assigner) {  
            return new WindowedStream(this, assigner);  
        }  
    ...  
}
```

## 10. KeyedStream#window()

```
@Public  
public class KeyedStream<T, KEY> extends DataStream<T> {  
    ...  
  
    @PublicEvolving  
    public <W extends Window> WindowedStream<T, KEY, W>  
        window(WindowAssigner<? super T, W> assigner) {  
            return new WindowedStream(this, assigner);  
        }  
    ...  
}
```

## 10. KeyedStream#window()

```
@Public  
public class KeyedStream<T, KEY> extends DataStream<T> {  
    ...  
  
    @PublicEvolving  
    public <W extends Window> WindowedStream<T, KEY, W>  
        window(WindowAssigner<? super T, W> assigner) {  
            return new WindowedStream(this, assigner);  
        }  
    ...  
}
```

## 10. WindowAll

- Type
  - DataStream->AllWindowedStream
- Groups all the stream events according to some characteristic  
e.g., the data that arrived within the last 5 seconds

## 10. 예제: WindowAll

dataStream

```
.windowAll(TumblingEventTimeWindows.of(Time.seconds(5)));
```

## 10. DataStream#windowAll()

```
@Public  
public class DataStream<T> {  
    ...  
  
    @PublicEvolving  
    public <W extends Window> AllWindowedStream<T, W>  
        windowAll(WindowAssigner<? super T, W> assigner) {  
            return new AllWindowedStream(this, assigner);  
        }  
    ...  
}
```

## 10. Example: Max and Min

- 주식 거래 회사에서 작업하고 있으며, 실시간으로 흐르는 주식 데이터를 분석하는 것이 우리의 업무입니다. 우리의 목표는 지정된 시간 윈도우 내에서 각 회사의 최고 및 최저 주식 가격을 찾는 것입니다.

## 10. Example: MaxBy and MinBy

- 이전 시나리오를 이어서 진행하면, 우리는 실시간 주식 데이터를 분석하고 있습니다. 이번에는 단순히 최고 및 최저 주식 가격을 찾는 것뿐만 아니라, 최고 및 최저 가격에 해당하는 전체 레코드를 검색하는 것이 목표입니다.  
이는 거래자에게 최고 또는 최저 가격이 발생한 시간과 같은 더 자세한 정보를 제공할 수 있습니다.

## 10. Example: Reduce

- 각 주식별로 각 윈도우 내에서 누적 거래량을 계산하고자 합니다.

이는 거래자가 어떤 주식이 많이 거래되고 가격 변동이 크게 일어날 수 있는지를 이해하는데  
중요할 수 있습니다.

## 10. Example: Aggregate

- 가격 패턴을 기반으로 의사 결정을 내리는 알고리즘 트레이딩 회사를 운영하고 있다고 가정해봅시다. 여기서 주목하는 패턴 중 하나는 갑작스러운 가격 상승입니다. 이를 위해 주어진 시간 윈도우 내에서의 최대 및 최소 주식 가격을 추적해야 합니다.

## 10. Example: Process

- 헤지 펀드를 운영하고 있으며 주식의 변동성에 관심이 있습니다.

변동성을 측정하는 한 가지 방법은 주어진 시간 윈도우 내에서 주식 가격의 표준 편차를 계산하는 것입니다.

# Chapter 3.

## 11. Tumbling Window

## 11. 예제: Tumbling Window

```
// tumbling event-time windows
input
    .keyBy(<key selector>)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    <windowed transformation>(<window function>);
// tumbling processing-time windows
input
    .keyBy(<key selector>)
    .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
    <windowed transformation>(<window function>);
```

## 11. 예제: Tumbling Window

```
// daily tumbling event-time windows offset by -8 hours.  
input  
    .keyBy(<key selector>)  
    .window(TumblingEventTimeWindows.of(Time.days(1), Time.hours(-8)))  
    <windowed transformation>(<window function>);
```

## 11. TumblingEventTimeWindows

```
@PublicEvolving
public class TumblingEventTimeWindows extends WindowAssigner<Object,
TimeWindow> {
    private static final long serialVersionUID = 1L;
    private final long size;
    private final long globalOffset;
    private Long staggerOffset = null;
    private final WindowStagger windowStagger;
```

## 11. TumblingEventTimeWindows.of()

```
public static TumblingEventTimeWindows of(Time size) {  
    return new TumblingEventTimeWindows(size.toMilliseconds(), 0,  
    WindowStagger.ALIGNED);  
}  
  
staggerOffset =  
windowStagger.getStaggerOffset(context.getCurrentProcessingTime(), size);
```

## 11. TumblingEventTimeWindows.of()

```
long start = TimeWindow.getWindowStartWithOffset(  
    timestamp, (globalOffset + staggerOffset) % size, size);
```

```
public static long getWindowStartWithOffset(long timestamp, long offset, long  
windowSize) {  
    final long remainder = (timestamp - offset) % windowSize;  
    if (remainder < 0) { return timestamp - (remainder + windowSize); }  
    else { return timestamp - remainder; }  
}
```

## 11. WindowStagger

```
@PublicEvolving
public enum WindowStagger {
    ALIGNED { ... return 0L; }

    RANDOM { ... return (long) (ThreadLocalRandom.current().nextDouble() * size); }

    NATURAL { ... return Math.max(0, currentProcessingTime -
        currentProcessingWindowStart); }

};
```

## 11. WindowOperator

```
@Override  
public Collection<TimeWindow> assignWindows(  
    Object element, long timestamp, WindowAssignerContext context) { ... }
```

```
@Internal  
public class WindowOperator<K, IN, ACC, OUT, W extends Window> ... {  
    @Override  
    public void processElement(StreamRecord<IN> element) throws Exception {  
        final Collection<W> elementWindows = windowAssigner.assignWindows(  
            element.getValue(), element.getTimestamp(), windowAssignerContext);  
        ... }
```

## 11. WindowOperator

```
@Override  
public Collection<TimeWindow> assignWindows(  
    Object element, long timestamp, WindowAssignerContext context) {  
    if (timestamp > Long.MIN_VALUE) { ... }  
    else { throw new RuntimeException( ... )}  
}
```

## 11. WindowOperator

```
@Override  
public Collection<TimeWindow> assignWindows(  
    Object element, long timestamp, WindowAssignerContext context) {  
    ...  
    if (staggerOffset == null) {  
        staggerOffset =  
            windowStagger.getStaggerOffset(context.getCurrentProcessingTime(), size);  
    }  
    ...  
}
```

## 11. WindowOperator

```
@Override  
public Collection<TimeWindow> assignWindows(  
    Object element, long timestamp, WindowAssignerContext context) {  
    ...  
    long start =  
        TimeWindow.getWindowStartWithOffset(  
            timestamp, (globalOffset + staggerOffset) % size, size);  
    return Collections.singletonList(new TimeWindow(start, start + size));  
    ...  
}
```

## 11. Example

- 주식 거래에서는 정기적인 간격으로 주식 데이터를 분석하여 추세나 이상 현상을 식별하는 것이 일반적입니다. 예를 들어, 주식 시장 분석가는 단기적인 가격 움직임을 이해하기 위해 5분 간격으로 가장 높은 주식 가격을 추적하고자 할 수 있습니다.

## 11. Practice

- 금융 시장에서는 투자자와 분석가들이 잠재적인 투자 기회나 시장의 추세를 파악하기 위해 특정 시간 범위 내에서 주식의 평균 가격을 추적하는 경우가 많습니다. 예를 들어, 금융 분석가는 하루 종일 10분 간격으로 주식의 평균 가격을 계산하는 것에 관심이 있을 수 있습니다. 이러한 계산은 고변동 기간을 파악하거나 주식을 매수 또는 매도하는 최적의 시기를 찾는데 유용할 수 있습니다.

# Chapter 3.

## 12. Sliding Window

## 12. 예제: Sliding Window

```
// sliding event-time windows
input
    .keyBy(<key selector>)
    .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
        .<windowed transformation>(<window function>);
// sliding processing-time windows
input
    .keyBy(<key selector>)
    .window(SlidingProcessingTimeWindows.of(Time.seconds(10),
Time.seconds(5)))
        .<windowed transformation>(<window function>);
```

## 12. SlidingEventTimeWindows

```
@PublicEvolving
public class SlidingEventTimeWindows extends WindowAssigner<Object,
TimeWindow> {
    private static final long serialVersionUID = 1L;
    private final long size;
    private final long slide;
    private final long offset;
```

## 12. SlidingEventTimeWindows.assignWindows()

```
@Override  
public Collection<TimeWindow> assignWindows(  
    Object element, long timestamp, WindowAssignerContext context) { ...  
    List<TimeWindow> windows = new ArrayList<>((int) (size / slide));  
    long lastStart = TimeWindow.getWindowStartWithOffset(timestamp, offset,  
    slide);  
    for (long start = lastStart; start > timestamp - size; start -= slide) {  
        windows.add(new TimeWindow(start, start + size));  
    }  
    return windows;  
}
```

## 12. SlidingEventTimeWindows.assignWindows()

```
@Override  
public Collection<TimeWindow> assignWindows(  
    Object element, long timestamp, WindowAssignerContext context) { ...  
    List<TimeWindow> windows = new ArrayList<>((int) (size / slide));  
    long lastStart = TimeWindow.getWindowStartWithOffset(timestamp, offset,  
    slide);  
    for (long start = lastStart; start > timestamp - size; start -= slide) {  
        windows.add(new TimeWindow(start, start + size));  
    }  
    return windows;  
    ... }
```

## 12. 예제: Sliding Window

```
// sliding processing-time windows offset by -8 hours
input
    .keyBy(<key selector>)
    .window(SlidingProcessingTimeWindows.of(Time.hours(12), Time.hours(1),
Time.hours(-8)))
    .<windowed transformation>(<window function>);
```

## 12. Offset

Without Offset

Window 1: [00:00 - 12:00]

Window 2: [01:00 - 13:00]

Window 3: [02:00 - 14:00]

...

Window 24: [23:00 - 11:00 next day]

At 6:05,

Window 7: [6:00 – 18:00)

With Offset (-8 hours)

Window 1: [16:00 (previous day) - 04:00)

Window 2: [17:00 (previous day) - 05:00)

Window 3: [18:00 (previous day) - 06:00)

...

Window 24: [15:00 - 03:00 next day)

At 6:05,

Window 7: [22:00 (previous day) - 10:00)

## 12. Example

- 증권 회사에서는 잠재적인 투자 기회를 파악하기 위해 각 주식의 실시간 평균 가격을 모니터링하고자 합니다. 그러나 단순히 고정된 간격으로 평균 가격을 살펴보는 대신, 회사는 슬라이딩 시간 윈도우 내의 평균 가격을 분석하고자 합니다.  
이렇게 함으로써 주식 가격 움직임에 대한 더 유연하고 실시간적인 이해를 얻을 수 있습니다.  
구체적으로는 1분마다 슬라이딩되는 5분 간격의 주식 평균 가격을 계산하고자 합니다.

## 12. Practice

- 헤지 펀드는 비정상적인 거래 활동을 식별하기 위해 각 주식의 실시간 거래량을 모니터링하고자 합니다. 특히, 거래 활동을 더 자세히 분석하기 위해 슬라이딩 시간 윈도우에서 총 거래량을 분석하고자 합니다.  
2분마다 슬라이딩되는 10분 간격의 총 거래량을 계산하고자 합니다.

# Chapter 3.

## 13. Session Window

## 13. 예제: Session Window

```
// event-time session windows with static gap
input
    .keyBy(<key selector>)
        .window(EventTimeSessionWindows.withGap(Time.minutes(10)))
            <windowed transformation>(<window function>);
// processing-time session windows with static gap
input
    .keyBy(<key selector>)
        .window(ProcessingTimeSessionWindows.withGap(Time.minutes(10)))
            <windowed transformation>(<window function>);
```

## 13. EventTimeSessionWindows.assignWindows()

```
public class EventTimeSessionWindows extends MergingWindowAssigner<Object,  
TimeWindow> {  
    ...  
    @Override  
    public Collection<TimeWindow> assignWindows(  
        Object element, long timestamp, WindowAssignerContext context) {  
        return Collections.singletonList(  
            new TimeWindow(timestamp, timestamp + sessionTimeout)); }  
    ... }
```

## 13. EventTimeSessionWindows

```
public class EventTimeSessionWindows extends  
MergingWindowAssigner<Object, TimeWindow> {  
    ...  
    @Override  
    public Collection<TimeWindow> assignWindows(  
        Object element, long timestamp, WindowAssignerContext context) {  
        return Collections.singletonList(  
            new TimeWindow(timestamp, timestamp + sessionTimeout)); }  
    ... }
```

## 13. MergingWindowAssigner

```
@Override  
@PublicEvolving  
public abstract class MergingWindowAssigner<T, W extends Window> extends  
WindowAssigner<T, W> {  
    private static final long serialVersionUID = 1L;  
    public abstract void mergeWindows(Collection<W> windows,  
        MergeCallback<W> callback);  
    public interface MergeCallback<W> {  
        void merge(Collection<W> toBeMerged, W mergeResult);  
    }  
}
```

## 13. MergingWindowAssigner

```
@Override  
@PublicEvolving  
public abstract class MergingWindowAssigner<T, W extends Window> extends  
WindowAssigner<T, W> {  
    private static final long serialVersionUID = 1L;  
    public abstract void mergeWindows(Collection<W> windows,  
        MergeCallback<W> callback);  
    public interface MergeCallback<W> {  
        void merge(Collection<W> toBeMerged, W mergeResult);  
    }  
}
```

## 13. TimeWindow.mergeWindows()

```
@PublicEvolving
public class TimeWindow extends Window {
    ...
    public static void mergeWindows(
        Collection<TimeWindow> windows,
        MergingWindowAssigner.MergeCallback<TimeWindow> c) {
        ...
        Collections.sort( ... );
        List<Tuple2<TimeWindow, Set<TimeWindow>>> merged =
            new ArrayList<>();
        ...
    } ...
}
```

## 13. TimeWindow.mergeWindows()

```
public static void mergeWindows(  
    Collection<TimeWindow> windows,  
    MergingWindowAssigner.MergeCallback<TimeWindow> c) {  
    ...  
    for (TimeWindow candidate : sortedWindows) {  
        if (currentMerge == null) { ...  
        } else if (currentMerge.f0.intersects(candidate)) { ...  
        } else { merged.add(currentMerge); ...  
        }  
    }  
    ... }  
}
```

## 13. 예제: Session Window

```
// event-time session windows with dynamic gap
input
    .keyBy(<key selector>)
        .window(EventTimeSessionWindows.withDynamicGap((element) -> { ... }))
            <windowed transformation>(<window function>);
// processing-time session windows with dynamic gap
input
    .keyBy(<key selector>)
        .window(ProcessingTimeSessionWindows.withDynamicGap((element) -> {
// determine and return session gap })
            <windowed transformation>(<window function>);
```

## 13. 예제: Dynamic Gap

```
stream
    .keyBy(value -> value)
    .window(EventTimeSessionWindows.withDynamicGap(new
SessionWindowTimeGapExtractor<String>() {
    @Override
    public long extract(String element) {
        return element.length() * 1000L;
    }
})) 
    .reduce((ReduceFunction<String>) (value1, value2) -> value1 + ", " + value2);
```

## 13. Example

- 금융 공학, 특히 파생 금융 분야에서는 옵션 가격 산출이 중요한 작업입니다.
- 옵션 가격 산출에서 주요한 요소 중 하나는 주로 기초 자산의 과거 가격에서 계산되는 변동성입니다. 예를 들어, 대규모 투자 은행에서 양적(Quantitative) 분석가로 일하고 있으며, 실시간으로 다양한 주식의 실현 변동성(Realized Volatility)을 계산하는 작업을 맡았습니다. 실현 변동성은 실현 분산(Realized Variance)의 제곱근이며, 실현 분산은 제곱된 수익률의 합입니다. 이 예시에서는 각 거래 세션의 실현 변동성을 계산합니다. 세션은 주식의 거래 활동 기간으로 정의되며, 활동 기간 사이에 비활동 기간 또는 캡이 있는 경우 (이 경우 15분 이상의 캡을 세션의 끝과 다음 세션의 시작으로 간주) 세션을 구분합니다.

## 13. Practice

- 헤지펀드에서 금융 분석가로 일하고 있으며, 실시간으로 특정 주식의 거래 활동을 분석하는 업무를 맡았습니다. 구체적으로는 각 거래 세션의 총 거래량을 계산하고자 합니다. 세션은 주식의 거래 활동 기간으로 정의되며, 활동 기간 사이에 비활동 기간 또는 갭이 있는 경우 (이 경우 15분 이상의 갭을 세션의 끝과 다음 세션의 시작으로 간주) 세션을 구분합니다.

이 작업을 위해 각 거래 세션의 총 거래량(거래량의 합)을 계산하는 Flink job을 구현해야 합니다. ProcessWindowFunction으로 구현하시길 바랍니다.

# Chapter 3.

## 14. Global Window

## 14. 예제: Global Window

```
input  
    .keyBy(<key selector>)  
    .window(GlobalWindows.create())  
    .<windowed transformation>(<window function>);
```

## 14. GlobalWindows.assignWindows()

```
@PublicEvolving
public class GlobalWindows extends WindowAssigner<Object, GlobalWindow> {
    ...
    @Override
    public Collection<GlobalWindow> assignWindows(
        Object element, long timestamp, WindowAssignerContext context) {
        return Collections.singletonList(GlobalWindow.get());
    }
    ...
}
```

## 14. GlobalWindow.get()

```
@PublicEvolving  
public class GlobalWindow extends Window {  
    ...  
    public static GlobalWindow get() {  
        return INSTANCE;  
    }  
    ...  
}
```

## 14. Example

- 고빈도 거래(high-frequency trading)에서는 거래가 밀리초 단위로 실행되며, 거래의 수가 실행 시간보다 중요한 경우가 많습니다.  
여러분의 작업은 매 1000건의 거래 후 각 주식에 대한 총 거래 금액을 계산하는 것입니다. 이를 통해 특정 기간 동안 어떤 주식이 가장 많이 거래되는지를 판단하는 데 도움이 될 수 있습니다.

## 14. Practice

- Value at Risk (VaR)는 투자의 위험을 측정하는 지표입니다. 일반적인 시장 조건에서 일정 기간(예: 하루) 동안 투자 포트폴리오가 얼마나 손실을 입을 수 있는지를 추정합니다. VaR은 일반적으로 금융 업계의 기업과 규제 기관에서 가능한 손실을 보호하기 위해 필요한 자산의 양을 파악하는 데 사용됩니다.  
이 시나리오에서는 포트폴리오 데이터를 스트리밍하는 투자 회사를 고려해봅시다. 각 포트폴리오에는 다양한 투자가 포함되어 있으며, 각 포트폴리오의 가치는 시장 변동에 따라 변동합니다. 회사는 모든 활성 포트폴리오에 대한 Value at Risk를 지속적으로 계산하고 지표의 값이 특정 지표보다 높을 때 출력하려고 합니다.  
또한 각 포트폴리오에는 유효 기간이 있습니다. 포트폴리오의 유효 기간이 지난 경우 계산에서 제외되어야 합니다.

# Chapter 3.

## 15. Joining

## 15. Types of Joins

- Window Joins
  - Tumbling Window Join
  - Sliding Window Join
  - Session Window Join
- Interval Join

## 15. Window Joins

- Joins elements of two streams that
  - Share common key
  - Lie in the same window
- Windows are defined by window assigner
- Windows are evaluated on elements from both streams

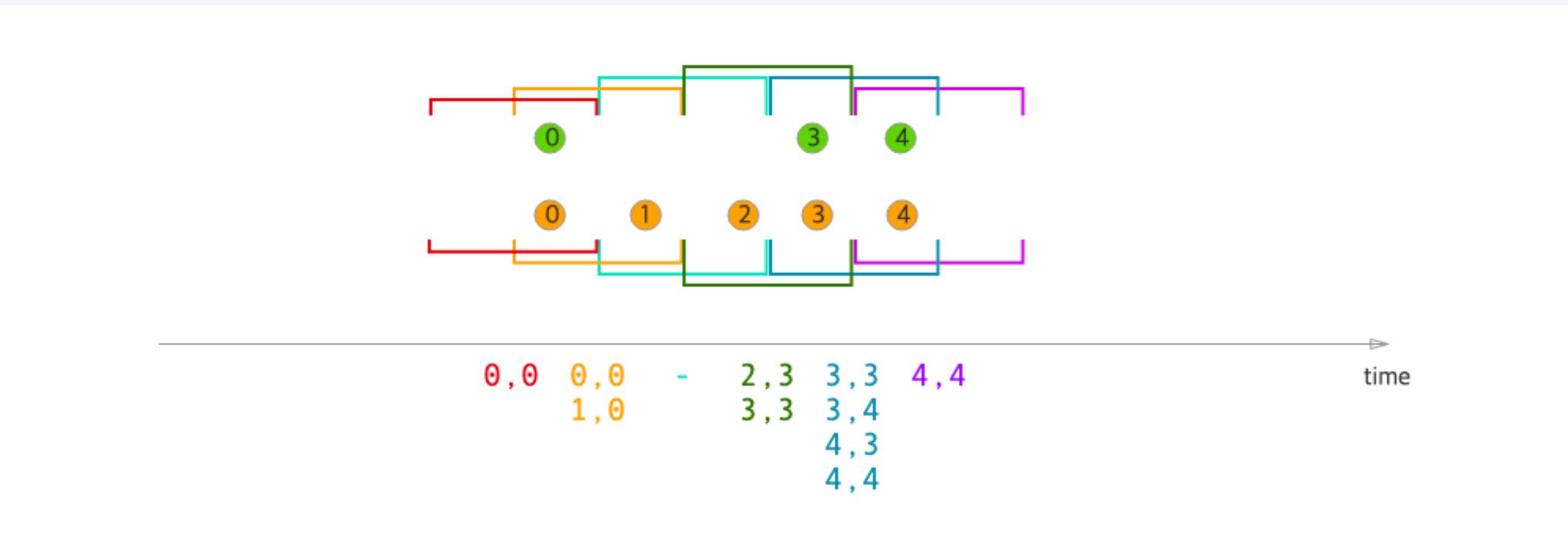
## 15. Window Joins

- Inner-join semantics
  - Elements from one stream will not be emitted if no match in other stream
- Largest Timestamp
  - Elements take largest timestamp that still lies in window
  - e.g. for window [5,10), join elements will have timestamp = 9

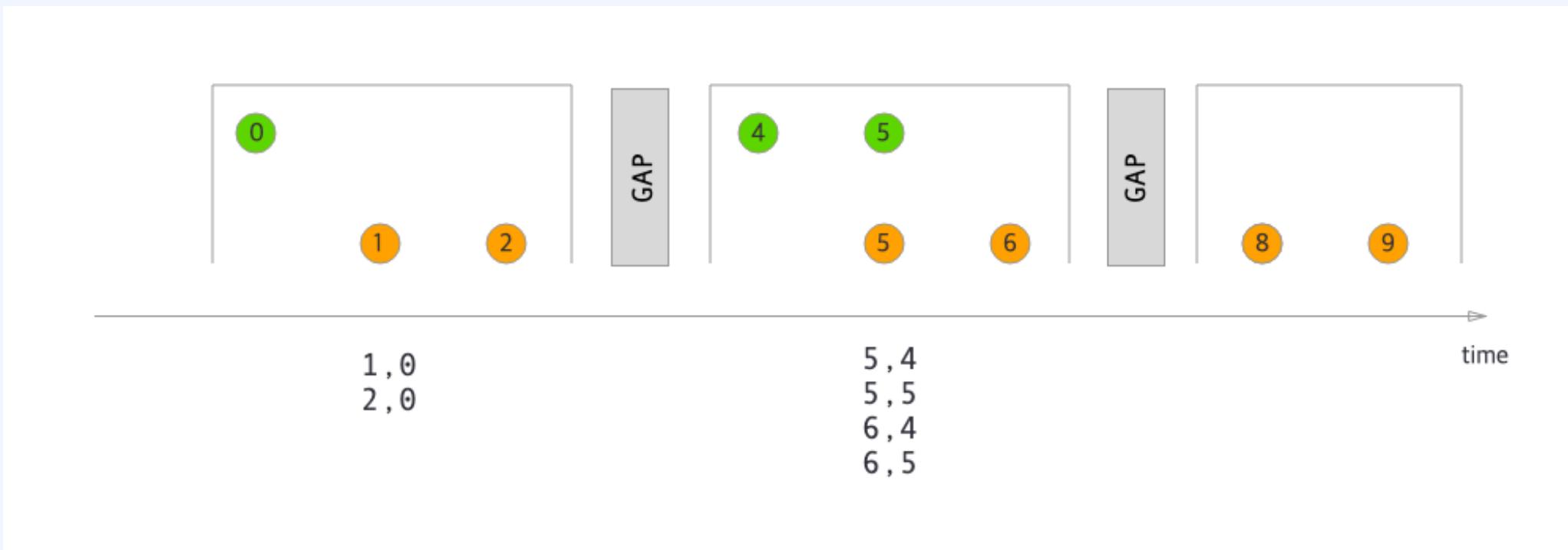
## 15. Tumbling Window Join



## 15. Sliding Window Join



## 15. Session Window Join



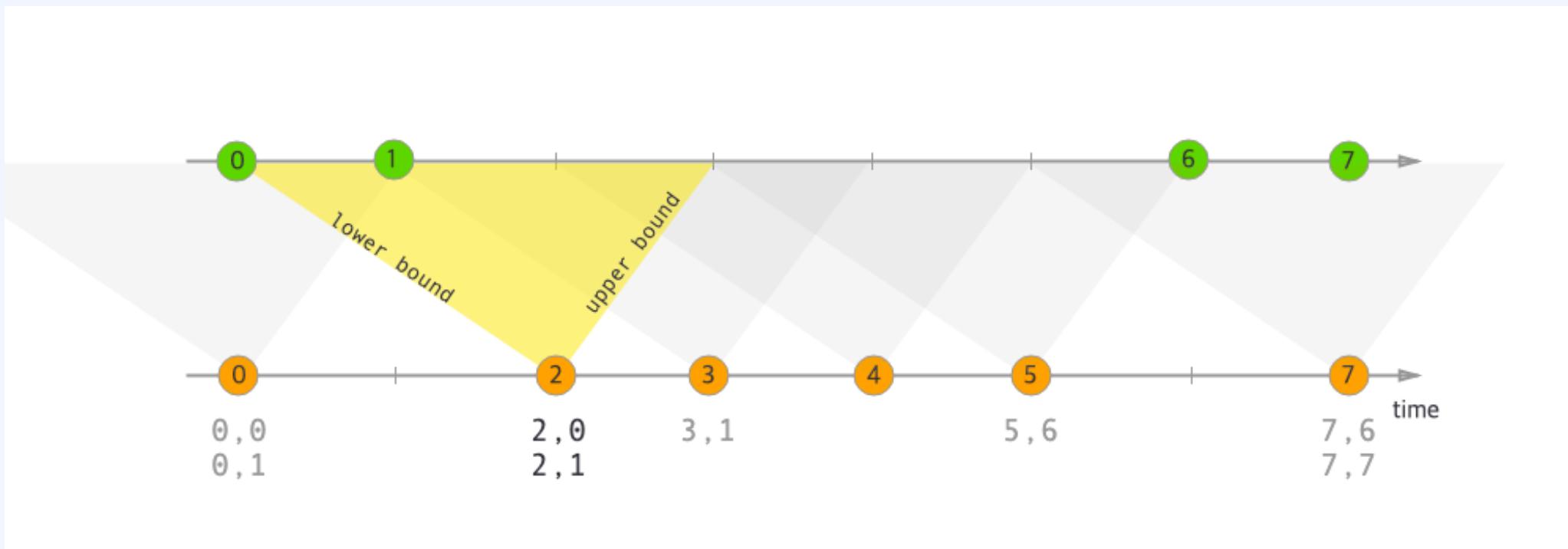
## 15. Interval Join

- Joins elements of two streams A and B that
  - A common key
  - Where elements of stream B have timestamps that lie in a relative time interval to those in stream A
- Expressed formally
$$a.timestamp + lowerBound \leq b.timestamp \leq a.timestamp + upperBound$$
  - $lowerBound \leq upperBound$  (can be either negative or positive)

## 15. Interval Join

- Interval Join only
  - supports event time
  - performs inner joins
- Larger Timestamp
  - assigned with the larger timestamp of the two elements.

## 15. 예제: Interval Join



## 15. 예제: Interval Join

```
orangeStream
    .keyBy(<KeySelector>)
    .intervalJoin(greenStream.keyBy(<KeySelector>))
    .between(Time.milliseconds(-2), Time.milliseconds(1))
    .process (new ProcessJoinFunction<Integer, Integer, String>(){
        @Override
        public void processElement(Integer left, Integer right, Context ctx,
        Collector<String> out) { out.collect(left + "," + right); } });
}
```

- orangeElem.ts + lowerBound <= greenElem.ts <= orangeElem.ts + upperBound

# Chapter 3.

## 16. Tumbling Window Join

## 16. 예제: Tumbling Window Join

```
orangeStream.join(greenStream)
    .where(<KeySelector>)
    .equalTo(<KeySelector>)
    .window(TumblingEventTimeWindows.of(Time.milliseconds(2)))
    .apply(new JoinFunction<Integer, Integer, String> () {
        @Override
        public String join(Integer first, Integer second) {
            return first + "," + second;
        }
    });
}
```

## 16. CoGroupedStreams.apply()

```
@Public  
public class CoGroupedStreams<T1, T2> {  
    ...  
    public <T> DataStream<T> apply(CoGroupFunction<T1, T2, T> function,  
        TypeInformation<T> resultType) {  
        ...  
    }  
    ...  
}
```

## 16. CoGroupedStreams.apply()

```
public <T> DataStream<T> apply(CoGroupFunction<T1, T2, T> function,  
TypeInformation<T> resultType) {  
    ...  
    function = input1.getExecutionEnvironment().clean(function);  
  
    UnionTypeInfo<T1, T2> unionType =  
        new UnionTypeInfo<>(input1.getType(), input2.getType());  
    UnionKeySelector<T1, T2, KEY> unionKeySelector =  
        new UnionKeySelector<>(keySelector1, keySelector2);  
    ...  
}
```

## 16. CoGroupedStreams.apply()

```
public <T> DataStream<T> apply(CoGroupFunction<T1, T2, T> function,  
TypeInformation<T> resultType) { ...  
SingleOutputStreamOperator<TaggedUnion<T1, T2>> taggedInput1 =  
    input1.map(new Input1Tagger<T1, T2>());  
taggedInput1.getTransformation().setParallelism(input1.getParallelism(), false);  
taggedInput1.returns(unionType);  
  
SingleOutputStreamOperator<TaggedUnion<T1, T2>> taggedInput2 =  
    input2.map(new Input2Tagger<T1, T2>());  
taggedInput2.getTransformation().setParallelism(input2.getParallelism(), false);  
taggedInput2.returns(unionType); ... }
```

## 16. CoGroupedStreams.apply()

```
public <T> DataStream<T> apply(CoGroupFunction<T1, T2, T> function,  
TypeInformation<T> resultType) {  
    ...  
  
    DataStream<TaggedUnion<T1, T2>> unionStream =  
        taggedInput1.union(taggedInput2);  
  
    windowedStream = new KeyedStream<TaggedUnion<T1, T2>, KEY>  
        (unionStream, unionKeySelector, keyType)  
            .window(windowAssigner);  
    ...  
}
```

## 16. CoGroupedStreams.apply()

```
public <T> DataStream<T> apply(CoGroupFunction<T1, T2, T> function,  
TypeInformation<T> resultType) { ...  
    if (trigger != null) {  
        windowedStream.trigger(trigger);  
    }  
    if (evictor != null) {  
        windowedStream.evictor(evictor);  
    }  
    if (allowedLateness != null) {  
        windowedStream.allowedLateness(allowedLateness);  
    } ... }
```

## 16. CoGroupedStreams.apply()

```
public <T> DataStream<T> apply(CoGroupFunction<T1, T2, T> function,  
TypeInformation<T> resultType) {  
    ...  
    return windowedStream.apply(  
        new CoGroupWindowFunction<T1, T2, T, KEY, W>(function), resultType);  
    ...  
}
```

## 16. WindowedStream.apply()

```
@Public  
public class WindowedStream<T, K, W extends Window> {  
    public <R> SingleOutputStreamOperator<R> apply(  
        WindowFunction<T, R, K, W> function, TypeInformation<R> resultType) {  
        ...  
        return input.transform(opName, resultType,  
            operator).setDescription(opDescription);  
        ...  
    }  
}
```

## 16. DataStream vs SingleOutputStreamOperator

- DataStream
- SingleOutputStreamOperator: Chaining
  - `stream.map(...).filter(...).reduce(...)`

## 16. Example

- 주식 거래에서 갑작스러운 거래량 증가는 종종 회사와 관련된 중요한 사건을 나타냅니다. 심볼과 윈도우 시간을 기반으로 NYSE와 NASDAQ 두 개의 주식 거래소의 스트림을 Tumbling Window Join을 사용하여 결합하겠습니다.  
결합된 스트림은 Window(Tumbling) 연산자에 의해 처리될 것입니다. 윈도우는 특정 시간 간격(예: 매 1분)을 기반으로 정의되며, 각 주식 심볼의 거래량 합계를 계산할 것입니다. 합계가 미리 정의된 임계값을 초과하면 거래량 급증이 발생한 것으로, 이는 이례적인 거래 활동을 나타냅니다.

## 16. Practice

- 주식 거래에서 흥미로운 시나리오 중 하나는 급격한 가격 하락을 탐지하는 것입니다. 이전 예제의 ProcessWindowFunction을 수정하여 총 거래량 대신 종가를 계산하겠습니다. 그런 다음 Window(Global) 연산자를 적용하여 일정 기간 (예: 거래일) 동안의 최고 가격을 계산해야 합니다.

종가는 최고 가격으로부터 일정 비율 (예: 10%) 이상 하락하는 경우, 급격한 가격 하락으로 간주됩니다. 해당 부분은 차후 CoFlatMapFunction 때 진행하겠습니다.

# Chapter 3.

## 17. Sliding Window Join

## 17. 예제: Sliding Window Join

```
orangeStream.join(greenStream)
    .where(<KeySelector>
        .equalTo(<KeySelector>
            .window(SlidingEventTimeWindows.of(Time.milliseconds(2) /* size */,
                Time.milliseconds(1) /* slide */))
            .apply(new JoinFunction<Integer, Integer, String> () {
                @Override
                public String join(Integer first, Integer second) {
                    return first + "," + second;
                }
            });
        );
```

## 17. Example

- 금융 분야에서 상관관계는 두 가지 증권이 서로 어떻게 움직이는지를 나타내는 통계적 측정 값입니다. 상관관계는 포트폴리오 관리에서 사용되며, 상관계수로 계산되며 값은 -1과 1 사이에 있어야 합니다.

예를 들어, 금과 은 같은 두 개의 상품이 있다고 가정해보겠습니다. 금과 은 가격은 역사적으로 양의 상관관계를 가지고 있습니다. 만약 일정한 슬라이딩 윈도우를 사용하여 실시간 상관계수 계산을 수행할 수 있다면, 이는 상품 거래자에게 이상 현상 또는 투자 기회에 대한 중요한 정보를 제공할 수 있을 것입니다.

이 예시에서는 금과 은 가격 사이의 상관관계를 모니터링하기 위해 슬라이딩 윈도우 조인을 사용합니다. 상관관계가 특정 임계값 아래로 떨어지면, 시스템은 추가 조사를 위한 알림을 생성합니다.

## 17. Practice

- 변동성은 투자 대상의 가치 변동 크기와 관련된 불확실성 또는 위험의 정도를 나타냅니다.

높은 변동성은 투자 대상의 가치가 더 넓은 범위로 퍼져있을 수 있다는 것을 의미합니다. 이는 투자 대상의 가격이 짧은 시간 동안에도 양 방향으로 급격하게 변동할 수 있음을 의미합니다. 반대로, 낮은 변동성은 투자 대상의 가치가 급격하게 변동하지 않으며, 시간에 따라 일정한 속도로 가치가 변화합니다.

두 가지 주식의 변동성을 비교하면 상대적인 위험 수준에 대한 유용한 통찰력을 제공할 수 있습니다. 이 과제에서는 슬라이딩 시간 윈도우 내에서 두 가지 다른 주식의 변동성을 비교하겠습니다. 그 후 변동성의 차이가 특정 임계값을 초과하면 추가 분석을 위한 알림을 생성하겠습니다.

# Chapter 3.

## 18. Session Window Join

## 18. 예제: Session Window Join

```
orangeStream.join(greenStream)
    .where(<KeySelector>
        .equalTo(<KeySelector>
            .window(EventTimeSessionWindows.withGap(Time.milliseconds(1)))
            .apply(new JoinFunction<Integer, Integer, String> () {
                @Override
                public String join(Integer first, Integer second) {
                    return first + "," + second;
                }
            });
        ));
```

## 18. Example

- 뉴스 기사의 발표가 회사의 주식 가격에 큰 영향을 미칠 수 있습니다. 예를 들어, 성공적인 제품 출시를 발표하는 기사는 주식 매수 열풍을 일으킬 수 있고, 금융 사건 또는 부정행위를 상세히 보도한 보고서는 주식 매도로 이어질 수 있습니다. 이 시나리오에서는 두 회사에 대한 뉴스 기사의 발표와 그들의 주식 거래 활동을 상관시킬 것입니다. 만약 다른 회사에 대한 뉴스 기사 발표 후에 특정 주식의 거래 활동이 자주 급증한다면, 두 회사 간에 상관 관계가 있을 수 있다는 것을 나타낼 수 있습니다.

Google의 제품 출시, 파트너십 또는 경영 결정과 관련된 어떠한 뉴스도 Microsoft의 주식 거래 활동의 급증으로 이어지는 경향이 있다고 가정하겠습니다. 이를 조사하기 위해, 우리는 Session Window Join을 사용하여 뉴스 기사의 게시 시간과 주식 거래 시간을 상관하여 분석하겠습니다.

## 18. Practice

- 전 세계 금융 시장은 서로 연결되어 있으며, 한 시장에서 발생하는 중요한 사건은 다른 시장에 영향을 미칠 수 있습니다. 예를 들어, 한 나라의 중요 지수가 크게 하락하면 다른 시장에서 매도가 발생할 수 있습니다. Session Window Join을 사용하여 여러 시장의 거래 세션을 비교함으로써 이러한 글로벌 트렌드를 파악할 수 있습니다.  
이 시나리오에서는 뉴욕 증권거래소(NYSE)와 런던 증권거래소(LSE)의 거래 활동을 모니터링하여 거래 세션의 겹침을 식별합니다. 이러한 겹침은 두 거래소에 상장된 주식에 영향을 미치는 교차 시장 사건을 나타낼 수 있습니다.

# Chapter 3.

## 19. Interval Join

## 19. 예제: Interval Join

```
orangeStream
    .keyBy(<KeySelector>)
    .intervalJoin(greenStream.keyBy(<KeySelector>))
    .between(Time.milliseconds(-2), Time.milliseconds(1))
    .process(new ProcessJoinFunction<Integer, Integer, String>(){
        @Override
        public void processElement(Integer left, Integer right, Context ctx,
        Collector<String> out) {
            out.collect(first + "," + second);
        }
    });
}
```

## 19.KeyedStream.intervalJoin()

```
@Public  
public class KeyedStream<T, KEY> extends DataStream<T> {  
    @PublicEvolving  
    public <T1> IntervalJoin<T, T1, KEY> intervalJoin(KeyedStream<T1, KEY>  
otherStream) {  
        return new IntervalJoin<>(this, otherStream);  
    }  
}
```

## 19. IntervalJoin.between()

```
@PublicEvolving
public static class IntervalJoin<T1, T2, KEY> {
    @PublicEvolving
    public IntervalJoined<T1, T2, KEY> between(Time lowerBound, Time
upperBound) {
        ...
        return new IntervalJoined<>(streamOne, streamTwo,
            lowerBound.toMilliseconds(), upperBound.toMilliseconds(),
            true, true);
        ...
    }
}
```

## 19. IntervalJoined.process()

```
@PublicEvolving
public static class IntervalJoined<IN1, IN2, KEY> {
    @PublicEvolving
    public <OUT> SingleOutputStreamOperator<OUT> process(
        ProcessJoinFunction<IN1, IN2, OUT> processJoinFunction,
        TypeInformation<OUT> outputType) {
        ...
    }
}
```

## 19. ProcessJoinFunction

```
@PublicEvolving
public abstract class ProcessJoinFunction<IN1, IN2, OUT> extends
AbstractRichFunction {
    ...
    public abstract void processElement(IN1 left, IN2 right, Context ctx,
Collector<OUT> out)
        throws Exception;
    ...
}
```

## 19. IntervalJoined.process()

```
@PublicEvolving
public <OUT> SingleOutputStreamOperator<OUT> process(
    ProcessJoinFunction<IN1, IN2, OUT> processJoinFunction,
    TypeInformation<OUT> outputType) {
    final ProcessJoinFunction<IN1, IN2, OUT> cleanedUdf =
        left.getExecutionEnvironment().clean(processJoinFunction);
    final IntervalJoinOperator<KEY, IN1, IN2, OUT> operator =
        new IntervalJoinOperator<>(..., cleanedUdf);
```

## 19. IntervalJoined.process()

```
@PublicEvolving
public <OUT> SingleOutputStreamOperator<OUT> process(
    ProcessJoinFunction<IN1, IN2, OUT> processJoinFunction,
    TypeInformation<OUT> outputType) {
    ...
    return left.connect(right)
        .keyBy(keySelector1, keySelector2)
        .transform("Interval Join", outputType, operator);
}
```

## 19. ConnectedStreams.transform()

```
@Public  
public class ConnectedStreams<IN1, IN2> {  
    @PublicEvolving  
    public <R> SingleOutputStreamOperator<R> transform(  
        String functionName,  
        TypeInformation<R> outTypeInfo,  
        TwoInputStreamOperator<IN1, IN2, R> operator) {  
        return doTransform(functionName, outTypeInfo,  
            SimpleOperatorFactory.of(operator));  
    }  
}
```

## 19. Example

- 신용등급 변경 이벤트가 특정 주식의 거래에 어떤 영향을 미치는지 분석하겠습니다.

인터벌 조인을 통해 특정 시간 범위 내에서만 이벤트를 결합하도록 하여 신용등급 변경이 주식 거래에 미치는 영향을 파악합니다.

## 19. Practice

- 온라인 거래 플랫폼은 사용자의 플랫폼 내 활동과 해당 사용자의 금융 거래를 상호 연관시킬 수 있습니다.

사용자가 거래 플랫폼에 로그인하고 다양한 활동을 수행하는 시나리오를 고려해보겠습니다. 이러한 활동은 로그인 이벤트, 주식 보기 이벤트, 시장 동향 분석 이벤트, 거래 이벤트 등의 다른 유형의 이벤트로 기록됩니다.

거래 플랫폼(trading platform)은 이러한 데이터를 실시간으로 분석하여 사용자 동작에 대한 통찰력을 얻을 수 있습니다. 이러한 통찰력을 얻기 위해서는 사용자 ID와 세션 ID를 기반으로 이러한 다양한 이벤트 스트림을 조인해야 합니다.

# Chapter 4.

## 20. Connect Operator

## 20. 예제: Connect

```
DataStream<Integer> someStream = //...
```

```
DataStream<String> otherStream = //...
```

```
ConnectedStreams<Integer, String> connectedStreams =  
    someStream.connect(otherStream);
```

## 20. DataStream.connect()

```
@Public  
public class DataStream<T> {  
    protected final StreamExecutionEnvironment environment;  
    ...  
    public <R> ConnectedStreams<T, R> connect(DataStream<R> dataStream) {  
        return new ConnectedStreams<>(environment, this, dataStream);  
    }  
    ...  
}
```

## 20. ConnectedStreams

```
public class ConnectedStreams<IN1, IN2> {  
    ...  
    protected ConnectedStreams(  
        StreamExecutionEnvironment env, DataStream<IN1> input1,  
        DataStream<IN2> input2) {  
        this.environment = requireNonNull(env);  
        this.inputStream1 = requireNonNull(input1);  
        this.inputStream2 = requireNonNull(input2);  
    }  
    ...  
}
```

## 20. Methods in ConnectedStreams

- keyBy()
- map()
- flatMap()
- process()
- transform()
- doTransfrom()

## 20. Methods in ConnectedStreams

- keyBy() -> ConnectedStream{}
- map() -> transform()
- flatMap() -> transform()
- process() -> transform()
- transform() -> doTransform()
- doTransfrom() -> SingleOutputStreamOperator{}

## 20. ConnectedStreams.keyBy()

```
public class ConnectedStreams<IN1, IN2> {…  
    public <KEY> ConnectedStreams<IN1, IN2> keyBy(  
        KeySelector<IN1, KEY> keySelector1,  
        KeySelector<IN2, KEY> keySelector2,  
        TypeInformation<KEY> keyType) {  
            return new ConnectedStreams<>(  
                environment,  
                inputStream1.keyBy(keySelector1, keyType),  
                inputStream2.keyBy(keySelector2, keyType));  
        } … }
```

## 20. ConnectedStreams.map()

```
public class ConnectedStreams<IN1, IN2> {  
    ...  
    public <R> SingleOutputStreamOperator<R> map(  
        CoMapFunction<IN1, IN2, R> coMapper, TypeInformation<R>  
        outputType) {  
        return transform("Co-Map", outputType, new  
        CoStreamMap<>(inputStream1.clean(coMapper)));  
    }  
    ...  
}
```

## 20. DataStream.clean()

```
@Public  
public class DataStream<T> {  
    ...  
  
    protected <F> F clean(F f) {  
        return getExecutionEnvironment().clean(f);  
    }  
    ...  
}
```

## 20. ConnectedStreams.flatMap()

```
public class ConnectedStreams<IN1, IN2> {  
    ...  
    public <R> SingleOutputStreamOperator<R> flatMap(  
        CoFlatMapper<IN1, IN2, R> coFlatMapper, TypeInformation<R>  
        outputType) {  
        return transform(  
            "Co-Flat Map", outputType,  
            new CoStreamFlatMap<>(inputStream1.clean(coFlatMapper)));  
    }  
    ...  
}
```

## 20. ConnectedStreams.process()

```
public class ConnectedStreams<IN1, IN2> {  
    @Internal  
    public <K, R> SingleOutputStreamOperator<R> process(  
        KeyedCoProcessFunction<K, IN1, IN2, R> keyedCoProcessFunction,  
        TypeInformation<R> outputType) {  
        ...  
        return transform("Co-Keyed-Process", outputType, operator);  
    }  
    ...  
}
```

## 20. ConnectedStreams.transform()

```
public class ConnectedStreams<IN1, IN2> {  
    ...  
    @PublicEvolving  
    public <R> SingleOutputStreamOperator<R> transform(  
        String functionName,  
        TypeInformation<R> outTypeInfo,  
        TwoInputStreamOperatorFactory<IN1, IN2, R> operatorFactory) {  
            return doTransform(functionName, outTypeInfo,  
                operatorFactory);  
        }  
    ... }
```

## 20. ConnectedStreams.keyBy()

```
public class ConnectedStreams<IN1, IN2> {  
    ...  
    private <R> SingleOutputStreamOperator<R> doTransform(  
        @SuppressWarnings({"unchecked", "rawtypes"}) { ...  
            SingleOutputStreamOperator<R> returnStream =  
                new SingleOutputStreamOperator(environment, transform);  
  
            getExecutionEnvironment().addOperator(transform);  
  
            return returnStream;  
        } }  
}
```

# Chapter 3.

## 21. CoMap

## 21. 예제: CoMap

```
connectedStreams.map(new CoMapFunction<Integer, String, Boolean>() {  
    @Override  
    public Boolean map1(Integer value) {  
        return true;  
    }  
    @Override  
    public Boolean map2(String value) {  
        return false;  
    }  
});
```

## 21. Exercise

- 금융 거래 시나리오를 가정해봅시다. 이 시나리오에서는 한 회사가 머신러닝 모델을 사용하여 특정 주식의 미래 가격을 예측합니다. 이 예측된 가격은 회사의 거래 결정에 활용될 수 있습니다. 회사는 예측된 가격을 실시간 거래 가격과 비교하여 예측 모델의 정확도를 평가하고 필요한 경우 조정하고자 합니다.

이 시나리오에서는 Stock과 StockPrediction라는 두 개의 데이터 스트림이 있다고 가정해 봅시다. 우리는 주식 ID를 기준으로 이 두 개의 스트림을 연결하고 CoMap 함수를 사용하여 실시간 가격과 예측 가격을 비교할 수 있습니다.

## 21. Practice

- 위험 관리와 사기 탐지와 관련된 시나리오를 생각해보겠습니다. 실시간 거래 시스템에서는 거래자와 거래 봇의 행동을 모니터링하여 사기나 위험한 행동을 나타낼 수 있는 이상한 활동을 감지하는 것이 일반적입니다.

예를 들어, 우리는 두 개의 데이터 스트림을 가지고 있을 수 있습니다. 하나는 거래 거래 내역 (StockTransaction 스트림)이고, 다른 하나는 머신러닝 모델이나 위험 엔진에 의해 생성된 위험 프로파일링 점수 (RiskScore 스트림)입니다. 위험 점수는 거래의 규모, 거래 빈도 및 거래자의 과거 행동과 같은 다양한 요소를 고려한 종합적인 점수일 수 있습니다.

여기서는 RiskScore를 정하는 모델이 결정되어 있다고 가정하고 계산하도록 하겠습니다.

# Chapter 3.

## 22. CoFlatMap

## 22. 예제: CoFlatMap

```
connectedStreams.flatMap(new CoFlatMapFunction<Integer, String, String>() {  
    @Override  
    public void flatMap1(Integer value, Collector<String> out) {  
        out.collect(value.toString());  
    }  
    @Override  
    public void flatMap2(String value, Collector<String> out) {  
        for (String word: value.split(" ")) {  
            out.collect(word);  
        } } );
```

## 22. CoFlatMapExercise

- 새로운 시나리오에서는 투자 회사에서 거래 봇을 운용하는 중심에 있습니다. 이 봇들은 "AAPL 주식 100주 매수" 또는 "GOOG 주식 50주 매도"와 같은 거래 지시를 내릴 수 있습니다.  
봇의 명령에는 유효시간과 해당 명령을 수행할 최대 가격 및 최소 가격이 기재되어 있습니다.

## 22. CoProcessExercise

- 금융 산업에서는 특정 주식에 대한 뉴스를 최신 상태로 유지하는 것이 매우 중요합니다. 뉴스 이벤트는 주식의 가격에 큰 영향을 미칠 수 있으며 거래 결정에 영향을 줄 수 있습니다. 따라서 거래 플랫폼들은 종종 사용자에게 뉴스 업데이트를 제공합니다. 이러한 업데이트는 시장에 대한 일반적인 뉴스일 수도 있으며 특정 주식과 관련된 특정 뉴스 일 수도 있습니다.  
우리의 시나리오에서는 뉴스의 중요성에 따라 주식 거래를 다양하게 분할해보도록 하겠습니다.

# Chapter 3.

## 23. Iterate

## 23. 예제:Iterate

```
IterativeStream<Long> iteration = initialStream.iterate();
DataStream<Long> iterationBody = iteration.map /*do something*/;
DataStream<Long> feedback = iterationBody.filter(new FilterFunction<Long>(){
    @Override
    public boolean filter(Long value) throws Exception {
        return value > 0;
    }
});
iteration.closeWith(feedback);
```

## 23. DataStream.iterate()

```
@Public  
public class DataStream<T> { ...  
  
    @PublicEvolving  
    public IterativeStream<T> iterate() {  
        return new IterativeStream<>(this, 0);  
    } ...  
  
    @PublicEvolving  
    public IterativeStream<T> iterate(long maxWaitTimeMillis) {  
        return new IterativeStream<>(this, maxWaitTimeMillis);  
    } ... }
```

For example, iterate(60000)

## 23. IterativeStream {}

```
@PublicEvolving
public class IterativeStream<T> extends SingleOutputStreamOperator<T> { ...
    protected IterativeStream(DataStream<T> dataStream, long maxWaitTime) {
        super(dataStream.getExecutionEnvironment(),
              new FeedbackTransformation<>(dataStream.getTransformation(),
              maxWaitTime));
        this.originalInput = dataStream;
        this.maxWaitTime = maxWaitTime;
        setBufferTimeout(dataStream.environment.getBufferTimeout());
    } ... }
```

## 23. IterativeStream {}

```
@PublicEvolving
public class IterativeStream<T> extends SingleOutputStreamOperator<T> { ...
    protected IterativeStream(DataStream<T> dataStream, long maxWaitTime) {
        super(dataStream.getExecutionEnvironment(),
              new FeedbackTransformation<>(dataStream.getTransformation(),
                                          maxWaitTime);
        this.originalInput = dataStream;
        this.maxWaitTime = maxWaitTime;
        setBufferTimeout(dataStream.environment.getBufferTimeout());
    } ... }
```

## 23. FeedbackTransformation {}

```
@Internal  
public class FeedbackTransformation<T> extends Transformation<T> {  
    private final Transformation<T> input;  
    private final List<Transformation<T>> feedbackEdges;  
    private final Long waitTime;  
  
    public FeedbackTransformation(Transformation<T> input, Long waitTime) {  
        super("Feedback", input.getOutputType(), input.getParallelism(), false);  
        this.input = input;  
        this.waitTime = waitTime;  
        this.feedbackEdges = Lists.newArrayList(); ... }
```

## 23. FeedbackTransformation.addFeedbackEdge()

```
@Internal
```

```
public class FeedbackTransformation<T> extends Transformation<T> {
```

```
...
```

```
    public void addFeedbackEdge(Transformation<T> transform) {
```

```
        if (transform.getParallelism() != this.getParallelism()) {
```

```
            throw new UnsupportedOperationException("...");
```

```
}
```

```
        feedbackEdges.add(transform);
```

```
}
```

```
...
```

```
}
```

## 23. FeedbackTransformation.getTransitivePredecessors()

```
@Internal  
public class FeedbackTransformation<T> extends Transformation<T> { ...  
  
    @Override  
    public List<Transformation<?>> getTransitivePredecessors() {  
        List<Transformation<?>> result = Lists.newArrayList();  
        result.add(this);  
        result.addAll(input.getTransitivePredecessors());  
        return result;  
    }  
    ...  
}
```

## 23. IterativeStream.closeWith()

```
@SuppressWarnings({"unchecked", "rawtypes"})
public DataStream<T> closeWith(DataStream<T> feedbackStream) {
    Collection<Transformation<?>> predecessors =
        feedbackStream.getTransformation().getTransitivePredecessors();

    if (!predecessors.contains(this.transformation)) {
        throw new UnsupportedOperationException(
            "Cannot close an iteration with a feedback DataStream that does not
            originate from said iteration.");
    }
    ...
}
```

## 23. IterativeStream.closeWith()

```
@SuppressWarnings({"unchecked", "rawtypes"})
public DataStream<T> closeWith(DataStream<T> feedbackStream) {
    ...
    ((FeedbackTransformation) getTransformation())
        .addFeedbackEdge(feedbackStream.getTransformation());
    ...
    return feedbackStream;
}
```

## 23. IterativeStream.withFeedbackType()

```
public <F> ConnectedIterativeStreams<T, F>
withFeedbackType(TypeInformation<F> feedbackType) {
    return new ConnectedIterativeStreams<>(originalInput, feedbackType,
maxWaitTime);
}
```

## 23. IterativeStream.withFeedbackType()

```
@PublicEvolving
public class IterativeStream<T> extends SingleOutputStreamOperator<T> {
    ...
    @Public
    public static class ConnectedIterativeStreams<I, F> extends
    ConnectedStreams<I, F> {
        ...
    }
}
```

## 23. ConnectedIterativeStream.keyBy()

```
private UnsupportedOperationException groupingException =  
    new UnsupportedOperationException(  
        "Cannot change the input partitioning of an"  
        + "iteration head directly. Apply the partitioning on the input and"  
        + "feedback streams instead.");  
  
@Override  
public <KEY> ConnectedStreams<I, F> keyBy(  
    KeySelector<I, KEY> keySelector1, KeySelector<F, KEY> keySelector2,  
    TypeInformation<KEY> keyType) {  
    throw groupingException;  
}
```

## 23. Exercise

- 지수 이동 평균 (EMA)은 트레이더와 분석가들이 금융 상품의 가격 트렌드를 시간에 따라 추적하는 데 사용하는 인기 있는 지표입니다. 단순 이동 평균과는 달리, EMA는 최근 가격에 더 많은 가중치를 부여합니다.  
이 예시에서는 iterative를 사용하여 EMA를 구현합니다.

## 23. Practice

- 주식의 일일 가격 움직임을 기반으로 서로 다른 주식들 사이의 상관 그래프를 구축하는 시스템이 있다고 가정하겠습니다. 각 노드는 주식을 나타내고, 각 간선은 두 주식 간의 상관 관계를 나타냅니다. 이제 각 주식의 PageRank를 iterative를 활용하여서 계산해보겠습니다.

# Chapter 4.

## 01. Keyed State

## 01. State {}

```
@PublicEvolving  
public interface State {  
    /** Removes the value mapped under the current key. */  
    void clear();  
}
```

## 01. StateDescriptor {}

```
@PublicEvolving
public abstract class StateDescriptor<S extends State, T> implements Serializable
{
    ...
    protected final String name;
    private final AtomicReference<TypeSerializer<T>> serializerAtomicReference =
        new AtomicReference<>();
    @Nullable private String queryableStateName;
    @Nonnull private StateTtlConfig ttlConfig = StateTtlConfig.DISABLED;
    ...
}
```

## 01. StateDescriptor.initializeSerializerUnlessSet()

```
@PublicEvolving
public abstract class StateDescriptor<S extends State, T> implements Serializable
{ ... public void initializeSerializerUnlessSet(ExecutionConfig executionConfig) {
    if (serializerAtomicReference.get() == null) {
        checkState(typeInfo != null, "no serializer and no type info");
        // try to instantiate and set the serializer
        TypeSerializer<T> serializer = typeInfo.createSerializer(executionConfig);
        // use cas to assure the singleton
        if (!serializerAtomicReference.compareAndSet(null, serializer)) {
            LOG.debug("Someone else beat us at initializing the serializer.");
        }
    }
}
```

## 01. StateDescriptor.setQueryable()

```
@PublicEvolving
public abstract class StateDescriptor<S extends State, T> implements Serializable
{ ... public void setQueryable(String queryableStateName) {
    Preconditions.checkNotNull(
        ttlConfig.getUpdateType() == StateTtlConfig.UpdateType.Disabled,
        "Queryable state is currently not supported with TTL");
    if (this.queryableStateName == null) {
        this.queryableStateName =
            Preconditions.checkNotNull(queryableStateName, "Registration name");
    } else {
        throw new IllegalStateException("Queryable state name already set"); } } ... }
```

## 01. StateDescriptor.enableTimeToLive()

```
@PublicEvolving
public abstract class StateDescriptor<S extends State, T> implements Serializable
{ ... public void enableTimeToLive(StateTtlConfig ttlConfig) {
    Preconditions.checkNotNull(ttlConfig);
    if (ttlConfig.isEnabled()) {
        Preconditions.checkArgument(
            queryableStateName == null,
            "Queryable state is currently not supported with TTL");
    }
    this.ttlConfig = ttlConfig;
} ... }
```

## 01. StateTtlConfig.UpdateType {}

```
@PublicEvolving
public class StateTtlConfig implements Serializable {
    ...
    public enum UpdateType {
        Disabled,
        OnCreateAndWrite,
        OnReadAndWrite
    }
    ...
}
```

## 01. StateTtlConfig.StateVisibility {}

```
@PublicEvolving
public class StateTtlConfig implements Serializable {
    ...
    public enum StateVisibility {
        ReturnExpiredIfNotCleanedUp,
        NeverReturnExpired
    }
    ...
}
```

## 01. StateTtlConfig.StateVisibility {}

```
StateTtlConfig ttlConfig = StateTtlConfig  
    .newBuilder(Time.seconds(1))  
    .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)  
    .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)  
    .build();
```

```
ValueStateDescriptor<String> stateDescriptor = new ValueStateDescriptor<>("text  
state", String.class);  
stateDescriptor.enableTimeToLive(ttlConfig);
```

## 01. StateTtlConfig.TtlTimeCharacteristic {}

```
@PublicEvolving
public class StateTtlConfig implements Serializable {
    ...
    public enum TtlTimeCharacteristic {
        ProcessingTime
    }
    ...
}
```

# Chapter 4.

## 02. ValueState

## 02. ValueState

```
@PublicEvolving  
public interface ValueState<T> extends State {  
    T value() throws IOException;  
    void update(T value) throws IOException;  
}
```

## 02. Exercise

- 주식 시장에서는 각 회사별로 거래된 주식의 누적 거래량을 추적하는 것이 중요합니다. 실시간으로 흐르는 거래 이벤트를 활용하여 Flink의 ValueState를 사용하여 누적 거래량을 저장하고 업데이트할 수 있습니다.

ValueState는 한 키와 한 operator에 대해 scope가 지정된 state로, 한 번에 하나의 키에 대한 fault tolerance state를 제공합니다. 각 키에 대해 값을 시간에 따라 개별적으로 누적하는 데 이상적입니다.

다음은 예시 시나리오입니다. 주식 ID와 거래된 주식 수량을 포함하는 거래 이벤트를 수신하고 있습니다. 우리의 작업은 각 주식 ID에 대해 거래된 주식 수량의 누적량을 계산하는 것입니다.

## 02. Practice

- 고빈도 거래(HFT) 회사들은 매우 빠른 속도로 대량의 주문을 실행합니다. 이 실습 과제에서는 HFT에 관여하는 각 주식 symbol에 대해 평균 거래 가치와 총 거래 횟수를 추적하는 실시간 거래 분석 시스템에 초점을 맞출 것입니다.  
시스템은 또한 특정 조건이 충족될 때, 예를 들어 평균 거래 가치가 특정 임계값을 초과하거나 시간 창 내의 거래 횟수가 한도를 초과하는 경우와 같이 알림을 생성해야 합니다. 이는 잠재적인 시장 조작이나 다른 의심스러운 활동을 나타낼 수 있습니다.

# Chapter 4.

## 03. ReducingState

## 03. ReducingState

```
@PublicEvolving  
public interface ReducingState<T> extends MergingState<T, T> {}
```

```
@PublicEvolving  
public interface MergingState<IN, OUT> extends AppendingState<IN, OUT> {}
```

```
@PublicEvolving  
public interface AppendingState<IN, OUT> extends State {  
    OUT get() throws Exception;  
    void add(IN value) throws Exception;  
}
```

## 03. Exercise

- 가끔은 특정 주식에 대한 갑작스런 거래 금액 증가는 중요한 시장 움직임이나 사건을 나타낼 수 있습니다. 이러한 경우 거래 알고리즘이나 시장 감시자들이 이러한 사건을 가능한 빨리 식별하는 것이 중요합니다.  
이 실습 과제에서는 ReducingState를 사용하여 특정 기간 동안 최대 거래 금액을 추적하고, 새로운 거래가 이 값을 특정 비율로 초과할 때 경고를 생성할 수 있습니다.

## 03. Practice

- 주식 시장에서 특정 기간 동안의 평균 주가를 이해하는 것은 해당 주식의 추세를 파악하는데 도움이 될 수 있습니다.  
이동평균과 크게 벗어나는 주식 가격의 변동은 해당 주식에 대해 비정상적인 시장 상황을 나타낼 수 있으며 투자 기회 또는 리스크 컨트롤을 위한 경보가 될 수 있습니다.  
ReducingState를 사용하여 주식 가격의 합계와 가격 개수를 추적하고, 이를 통해 실시간으로 평균 주가를 계산할 수 있습니다.

# Chapter 4.

## 04. ListState

## 04. ListState

```
@PublicEvolving
public interface ListState<T> extends MergingState<T, Iterable<T>> {
    void update(List<T> values) throws Exception;
    void addAll(List<T> values) throws Exception;
}
```

## 04. Exercise

- 거래 분석에서는 이동 평균과 같은 분석을 위해 종종 주식의 과거 N개의 가격을 추적하는 것이 유용합니다. 그러나 계속해서 커지는 가격 기록을 유지하는 것은 메모리 효율적이지 않을 수 있습니다. 대신 ListState를 사용하여 각 주식의 마지막 N개의 가격을 리스트에 저장하고, 새로운 거래가 발생할 때마다 이를 분석하여 이동 평균을 계산할 수 있습니다. KeyedProcessFunction을 구현해 보겠습니다. 이 함수는 ListState를 사용하여 각 주식의 마지막 N개의 가격을 저장하고, 새로운 거래가 발생할 때마다 이동 평균을 계산합니다.

## 04. Practice

- 이 시나리오에서는 고순자산층(고액 투자자)을 대상으로 하는 투자 관리 시스템을 상상해보겠습니다. 각 투자자는 주식, 채권 또는 암호화폐와 같은 다양한 자산으로 구성된 포트폴리오를 가질 수 있습니다.

여러분의 애플리케이션은 자산 가격의 스트림을 받아와서 해당 포트폴리오를 업데이트합니다. 시스템은 투자 분석을 위해 간단한 기술적 지표를 계산하기 위해 각 포트폴리오 내의 각 자산의 최근 다섯 가격을 추적해야 합니다. 이 기술적 지표에는 최근 다섯 차례의 평균 가격과 같은 것들이 포함됩니다.

투자 관리자들은 이 정보를 활용하여 투자 결정을 내릴 수 있습니다.

# Chapter 4.

## 05. AggregatingState

## 05. AggregatingState

```
@PublicEvolving  
public interface AggregatingState<IN, OUT> extends MergingState<IN, OUT> {}
```

```
@PublicEvolving  
public interface MergingState<IN, OUT> extends AppendingState<IN, OUT> {}
```

```
@PublicEvolving  
public interface AppendingState<IN, OUT> extends State {  
    OUT get() throws Exception;  
    void add(IN value) throws Exception;  
}
```

## 05. Exercise

- 고려해야 할 것은 다양한 포트폴리오에 대해 수행된 지속적인 거래 스트림을 받는 실시간 포트폴리오 관리 시스템입니다. 각 거래에는 해당 포트폴리오에 속하는 정보, 매수/매도한 자산, 금액 및 단위당 가격과 같은 정보가 포함되어 있습니다.  
포트폴리오 관리에서 중요한 지표 중 하나는 가치 대 비용 비율입니다. 이는 포트폴리오의 현재 총 가치를 포트폴리오 구성에 필요한 총 비용으로 나눈 값입니다. 비용은 모든 매수에 대한 단위당 가격과 금액을 합한 것이고, 가치는 동일한 계산을 하되 현재 단위당 가격을 사용합니다.  
이 비율을 계산하는 것은 새로운 거래가 발생할 때마다 총 비용과 총 가치를 점진적으로 업데이트하는 것으로 최적화될 수 있으며, 이는 AggregatingState와 완벽히 어울립니다.

## 05. Practice

- 급변하는 금융 거래 세계에서 다른 고객들은 주식, 채권, 통화, 상품 등과 같은 자산을 거래합니다. 금융 기관에게는 각 고객의 순이익을 추적하는 것이 중요합니다. 이는 고객 프로파일링, 리스크 관리, 그리고 체계적인 비즈니스 결정을 위해 필수적입니다. 순이익은 자산 매수에 소요된 총 비용을 자산 매도로부터 얻은 총 수익에서 차감하여 계산됩니다.  
각 이벤트에는 자산 거래의 세부 정보 (자산 유형, 가격, 수량, 매수 또는 매도 여부)와 함께 고객 ID가 포함됩니다. 새로운 거래 이벤트가 처리됨에 따라 순이익이 갱신되고 방출되므로, 실시간으로 고객별 순이익을 확인할 수 있습니다.  
이러한 처리는 실시간으로 높은 수익을 창출하는 고객을 탐지하거나, 수익 임계값에 기반하여 경고를 트리거하거나, 리포트와 대시보드를 위해 실시간으로 수익 메트릭을 계산하는 등 여러 시나리오에서 유용합니다.

# Chapter 4.

## 06. MapState

## 06. MapState

```
@PublicEvolving
public interface MapState<UK, UV> extends State {
    UV get(UK key) throws Exception;
    void put(UK key, UV value) throws Exception;
    void putAll(Map<UK, UV> map) throws Exception;
    void remove(UK key) throws Exception;
    boolean contains(UK key) throws Exception;
    ...
}
```

## 06. MapState

```
@PublicEvolving
public interface MapState<UK, UV> extends State {
    ...
    Iterable<Map.Entry<UK, UV>> entries() throws Exception;
    Iterable<UK> keys() throws Exception;
    Iterable<UV> values() throws Exception;
    Iterator<Map.Entry<UK, UV>> iterator() throws Exception;
    boolean isEmpty() throws Exception;
}
```

## 06. Exercise

- 금융 세계에서는 자산 관리 회사들이 여러 고객이 다양한 자산을 거래하는 상황에 직면합니다. 실시간 거래 이벤트(매수/매도)와 자산 가격 업데이트에 따라 각 고객의 포트폴리오를 지속적으로 추적하고 조정해야 합니다. 또한 현재 시장 가격을 기준으로 각 자산의 손익 상태를 고객 포트폴리오에서 추적해야 합니다.  
매도 작업인 경우 포트폴리오를 업데이트하기 전에 고객이 해당 자산의 충분한 수량을 가지고 있는지 확인합니다. 매수 작업인 경우 자산의 수량과 비용을 간단히 포트폴리오에 추가합니다.

각 가격 업데이트를 처리하여 Broadcast state를 최신 가격으로 업데이트한 후 각 고객 포트폴리오의 자산 손익 상태를 확인합니다. 현재 가치(수량 \* 최신 가격)가 총 비용보다 크면 자산은 이익이 있는 것으로 간주하고, 그렇지 않으면 손실인 것으로 판단합니다.

## 06. Practice

- 금융 분야에서 자산 거래를 할 때 위험 관리는 가장 중요한 측면 중 하나입니다. 금융 기관과 자산 관리 회사들은 각 자산에 관련된 위험을 평가하고 위험 가중치를 할당합니다. 이러한 위험 가중치는 시장 조건 및 기타 요소에 따라 실시간으로 업데이트될 수 있습니다. 이러한 위험 가중치는 포트폴리오의 전반적인 위험을 계산하는 데 중요하며 의사 결정 과정에서 사용됩니다.

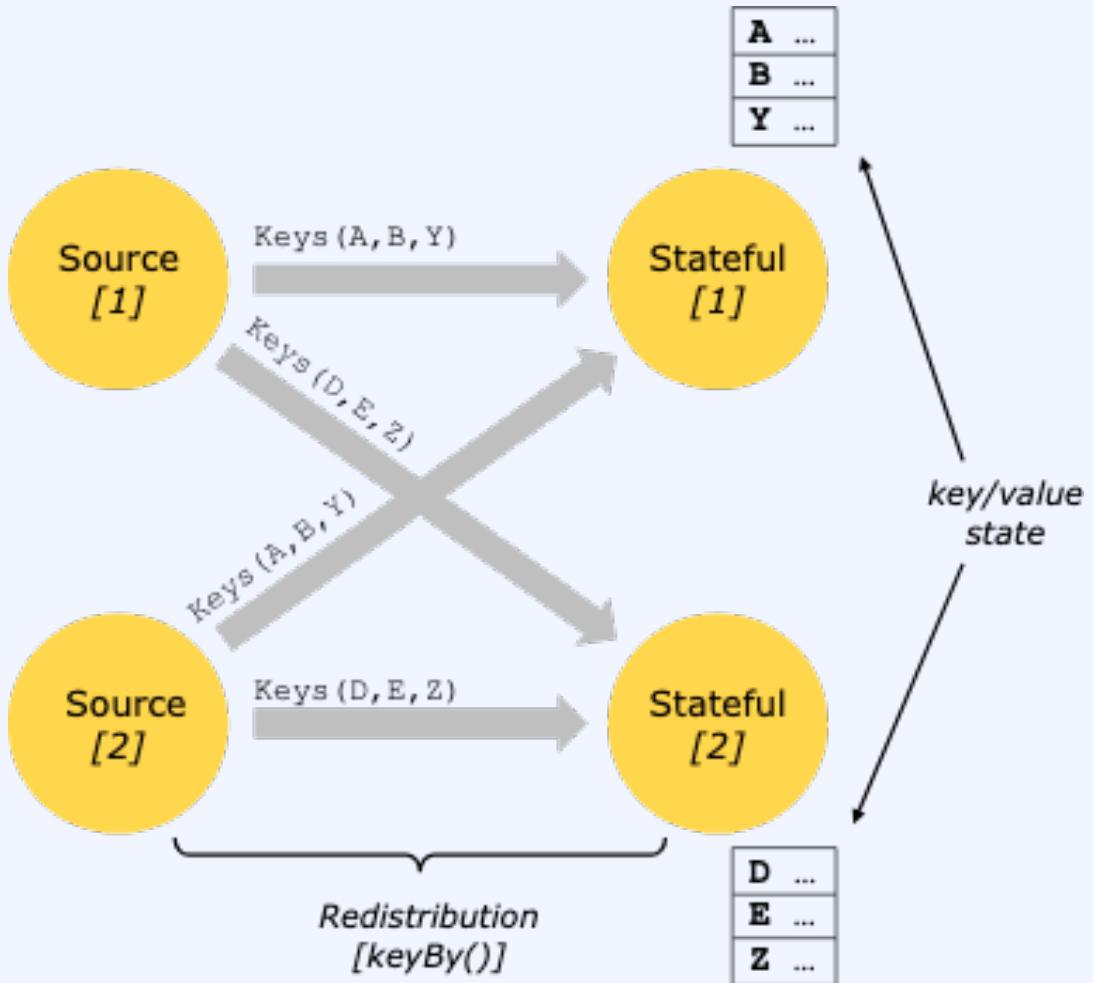
# Chapter 4.

## 07. Broadcast State 패턴

## 07. Recap

- Operations in a dataflow
  - one individual event at a time: event parser
  - remember information across multiple events: window operators  
-> **stateful**
- Fault tolerant
  - Checkpoints & Savepoints
- Queryable state
  - access state from outside of Flink during runtime
- State backends
  - HashMapStateBackend
  - EmbeddedRocksDBStateBackend

## 07. Recap (Keyed State)



## 07. Recap (Broadcast State)

- Broadcast State is a special type of Operator State
- Support use cases where records of one stream need to **be broadcasted to all downstream tasks**, where they are used to maintain *the same state* among all subtasks.
- Differs from other Operator State
  - Map format
  - specific operators that have as inputs a broadcasted stream and a non-broadcasted one
  - have *multiple broadcast states* with different names

## 07. Provided API

- Non-broadcast Stream
  - type Item with a Color and a Shape property
- Broadcast Stream
  - the Rules

## 07. Provided API

```
// key the items by color  
KeyedStream<Item, Color> colorPartitionedStream =  
    itemStream .keyBy(new KeySelector<Item, Color>() {...});
```

## 07. Provided API

```
// a map descriptor to store the name of the rule (string) and the rule itself.  
MapStateDescriptor<String, Rule> ruleStateDescriptor =  
    new MapStateDescriptor<>("RulesBroadcastState",  
        BasicTypeInfo.STRING_TYPE_INFO,  
        TypeInformation.of(new TypeHint<Rule>() {}));  
  
// broadcast the rules and create the broadcast state  
BroadcastStream<Rule> ruleBroadcastStream =  
    ruleStream.broadcast(ruleStateDescriptor);
```

## 07. Provided API

```
DataStream<String> output = colorPartitionedStream
    .connect(ruleBroadcastStream)
    .process(
        // type arguments in our KeyedBroadcastProcessFunction represent:
        // 1. the key of the keyed stream
        // 2. the type of elements in the non-broadcast side
        // 3. the type of elements in the broadcast side
        // 4. the type of the result, here a string
        new KeyedBroadcastProcessFunction<Color, Item, Rule, String>() {
            // my matching logic
        });
    }
```

## 07. Provided API

- If keyed: KeyedBroadcastProcessFunction
- If non-keyed: BroadcastProcessFunction

## 07. BroadcastProcessFunction and KeyedBroadcastProcessFunction

```
public abstract class BroadcastProcessFunction<IN1, IN2, OUT> extends  
BaseBroadcastProcessFunction {  
    public abstract void processElement(IN1 value, ReadOnlyContext ctx,  
Collector<OUT> out) throws Exception;  
    public abstract void processBroadcastElement(IN2 value, Context ctx,  
Collector<OUT> out) throws Exception;  
}
```

## 07. BroadcastProcessFunction and KeyedBroadcastProcessFunction

```
public abstract class KeyedBroadcastProcessFunction<KS, IN1, IN2, OUT> {  
    public abstract void processElement(IN1 value, ReadOnlyContext ctx,  
    Collector<OUT> out) throws Exception;  
    public abstract void processBroadcastElement(IN2 value, Context ctx,  
    Collector<OUT> out) throws Exception;  
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<OUT> out)  
    throws Exception;  
}
```

## 07. BroadcastProcessFunction and KeyedBroadcastProcessFunction

```
public abstract class KeyedBroadcastProcessFunction<KS, IN1, IN2, OUT> {  
    public abstract void processElement(IN1 value, ReadOnlyContext ctx,  
    Collector<OUT> out) throws Exception;  
    public abstract void processBroadcastElement(IN2 value, Context ctx,  
    Collector<OUT> out) throws Exception;  
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<OUT> out)  
    throws Exception;  
}
```

## 07. BroadcastProcessFunction and KeyedBroadcastProcessFunction

- get
  - ctx.getBroadcastState(MapStateDescriptor<K, V> stateDescriptor)
  - ctx.timestamp(),
  - ctx.currentWatermark()
  - ctx.currentProcessingTime()
- common
  - ctx.output(OutputTag<X> outputTag, X value).

## 07. BroadcastProcessFunction and KeyedBroadcastProcessFunction

- Only ctx in processBroadcastElement()
  - ctx.getBroadcastState(ruleStateDescriptor).put(value.name, value);

## 07. BroadcastProcessFunction and KeyedBroadcastProcessFunction

```
public abstract class KeyedBroadcastProcessFunction<KS, IN1, IN2, OUT> {  
    public abstract void processElement(IN1 value, ReadOnlyContext ctx,  
    Collector<OUT> out) throws Exception;  
    public abstract void processBroadcastElement(IN2 value, Context ctx,  
    Collector<OUT> out) throws Exception;  
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<OUT> out)  
    throws Exception;  
}
```

## 07. BroadcastProcessFunction and KeyedBroadcastProcessFunction

- Context in the processBroadcastElement()
  - applyToKeyedState(StateDescriptor<S, VS> stateDescriptor, KeyedStateFunction<KS, S> function)
    - register a KeyedStateFunction to be applied to all states of all **keys** associated with the provided *stateDescriptor*

## 07. Important Considerations

- There is no cross-task communication
- Order of events in Broadcast State may differ across tasks
- All tasks checkpoint their broadcast state
  - Hotspots <-> Parallelism(Size)
- No RocksDB state backend

# Chapter 4.

## 08. Checkpointing / Savepoints

## 08. State Persistence

- Fault tolerance
  - stream replay
  - checkpointing
    - a specific point in each of the input streams
    - along with the corresponding state for each of the operators
- Checkpointing interval
  - Overhead <-> Recovery time
- Snapshot: -> distributed file system

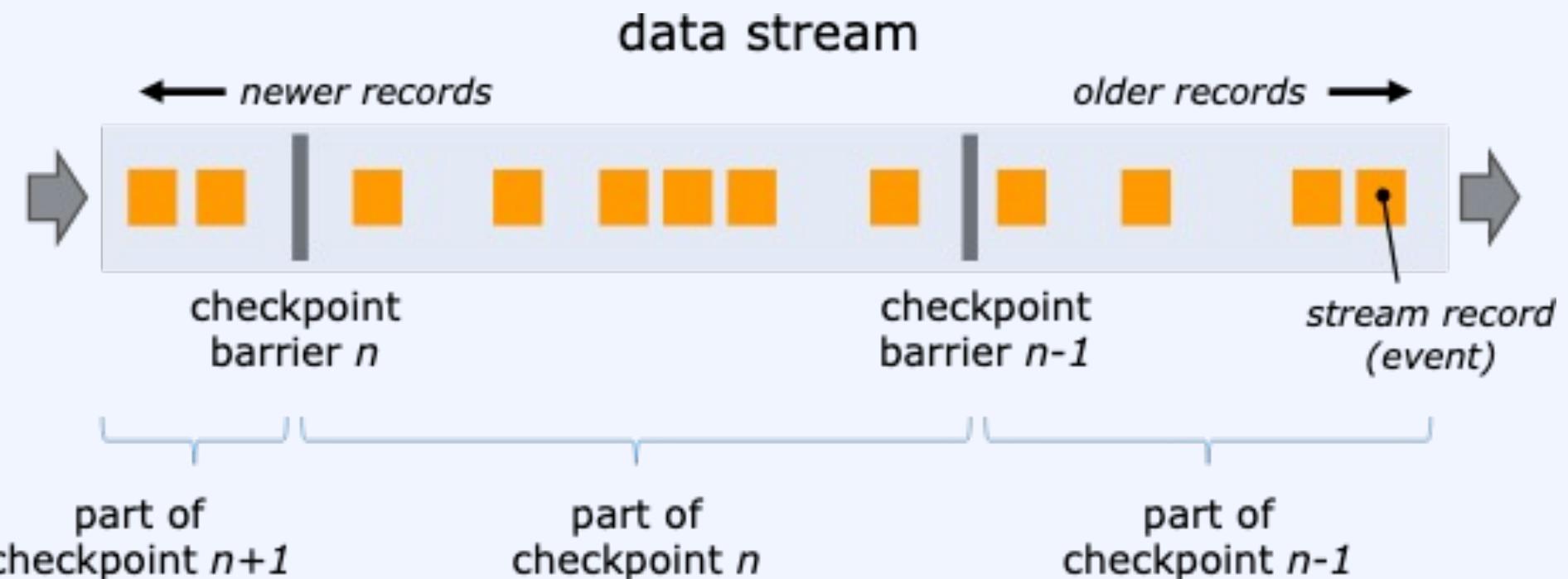
## 08. State Persistence

- In case of program failure
  - stops the distributed streaming dataflow
    - > restarts the operators
    - > resets them(operators) to the latest successful checkpoint
    - > input streams are reset to the point of the state snapshot

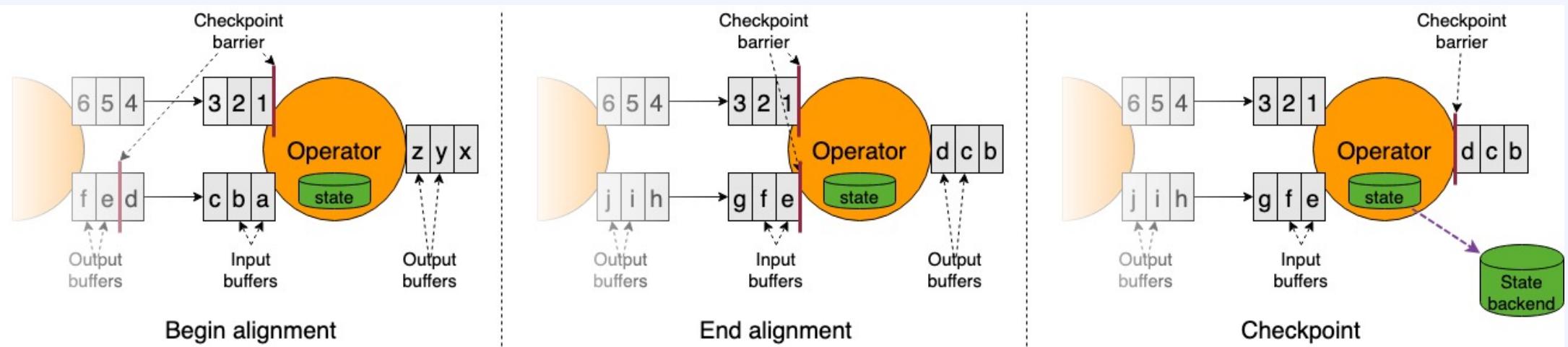
## 08. Checkpointing

- [“Lightweight Asynchronous Snapshots for Distributed Dataflows”](#)
  - checkpointing can be done asynchronously
- Since Flink 1.11
  - with or without alignment

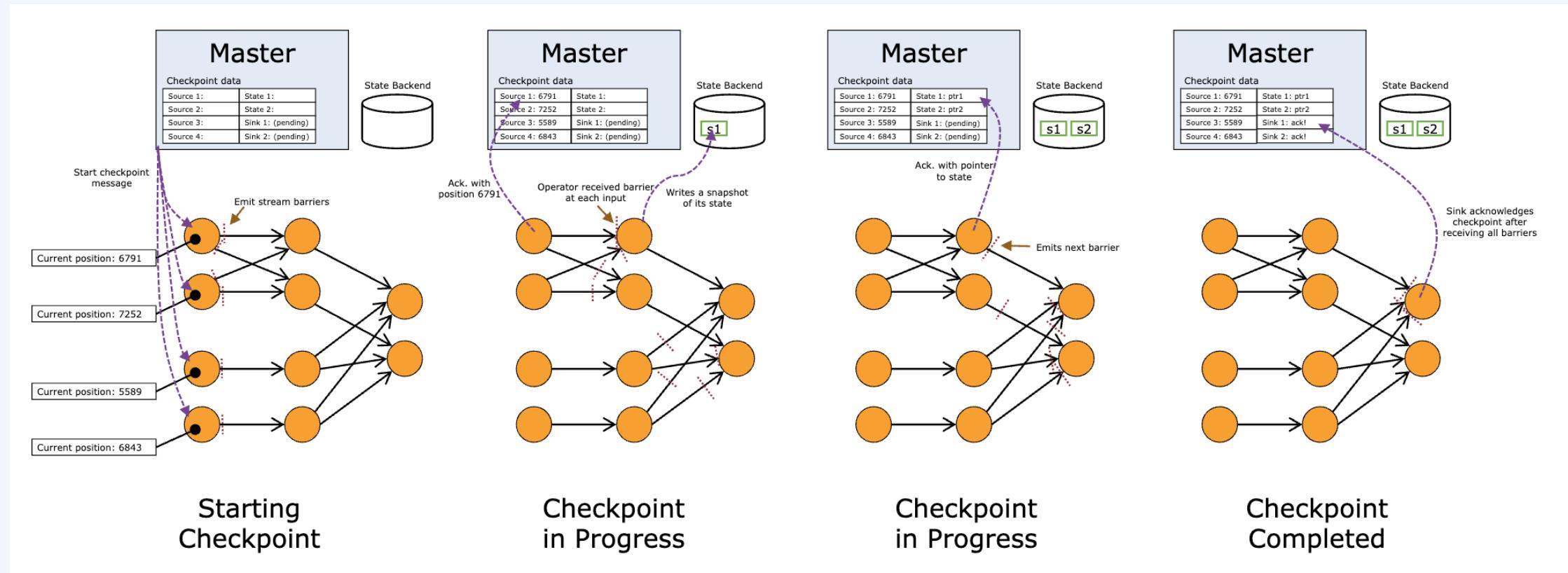
## 08. Checkpointing - Barriers



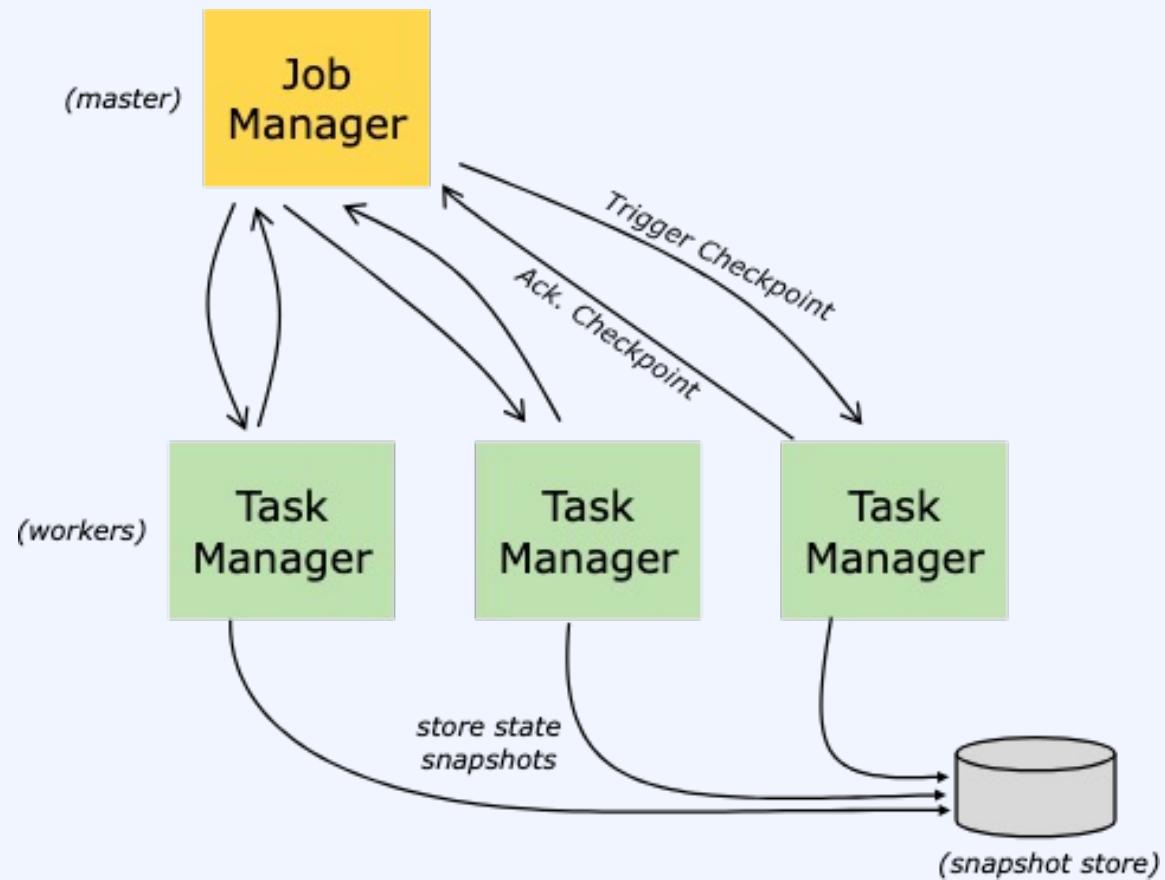
## 08. Checkpointing - Barriers



# 08. Checkpointing - Snapshotting Operator State



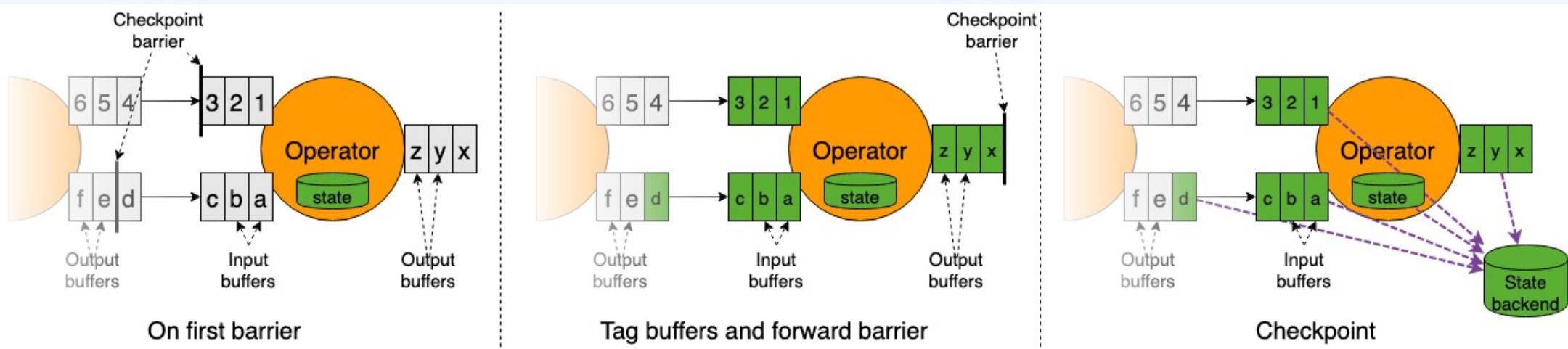
## 08. State Backends



## 08. Checkpointing - Recovery

- selects the latest completed checkpoint  $k$ .
- re-deploys the entire distributed dataflow,  
gives each operator the state
- sources are set to start reading the stream from position  $S_k$ .
  - In Apache Kafka, the consumer to start fetching from offset  $S_k$ .

## 08. Unaligned Checkpointing



## 08. Checkpointing – Unaligned Recovery

- recover the in-flight data
- starting processing any data from upstream operators

## 08. Savepoints

- manually triggered checkpoints
  - triggered by the user
  - don't automatically expire
- savepoints will always be aligned

## 08. Savepoints

- a directory with binary files
- meta data file

## 08. Savepoints – Assigning Operator IDs

```
DataStream<String> stream = env.  
    // Stateful source (e.g. Kafka) with ID  
    .addSource(new StatefulSource())  
    .uid("source-id") // ID for the source operator  
    .shuffle() // Stateful mapper with ID  
    .map(new StatefulMapper())  
    .uid("mapper-id") // ID for the mapper  
    // Stateless printing sink .print();  
    // Auto-generated ID
```

## 08. Savepoints – CLI

- Create
  - `./bin/flink savepoint <jobId> [savepointDirectory]`
- Run
  - `./bin/flink run -s [savepointPath] ...`
- Delete
  - `./bin/flink savepoint -d [savepointPath]`

## 08. Savepoints – Format

- Canonical Savepoint: a format that has been unified across all state backends
- Native Savepoint: a snapshot in the format specific for the used state backend

# 08. Checkpoints vs Savepoints

Operation	Canonical Savepoint	Native Savepoint	Aligned Checkpoint	Unaligned Checkpoint
State backend change	✓	x	x	x
State Processor API (writing)	✓	x	x	x
State Processor API (reading)	✓	!	!	x
Self-contained and relocatable	✓	✓	x	x
Schema evolution	✓	!	!	!
Arbitrary job upgrade	✓	✓	✓	x
Non-arbitrary job upgrade	✓	✓	✓	✓
Flink minor version upgrade	✓	✓	✓	x
Flink bug/patch version upgrade	✓	✓	✓	✓
Rescaling	✓	✓	✓	✓

# Chapter 4.

## 09. Queryable State

## 09. Queryable State

- Queryable State?
  - (Keyed) State to the outside world
  - eliminates the need for distributed operations/transactions\
  - useful for debugging purposes
- When querying a state object
  - (Without synchronization or copying) accessed from a concurrent thread
- No guarantees made about stability of the provided interfaces

## 09. Architecture

- QueryableStateClient: Flink 클러스터 외부에서 실행될 수 있으며 사용자 쿼리를 제출
- QueryableStateClientProxy: 각 TaskManager에서 실행되며, 클라이언트의 쿼리를 받아, 요청된 상태를 담당 TaskManager에서 가져와 클라이언트에게 반환하는 역할
- QueryableStateServer: 각 TaskManager에서 실행되며, 로컬에 저장된 상태를 제공하는 역할

## 09. Activating Queryable State

- Copy the flink-queryable-state-runtime-1.18-SNAPSHOT.jar from the opt/ folder of your Flink distribution, to the lib/ folder.
- Set the property queryable-state.enable to true.

## 09. Making State Queryable

- Copy the flink-queryable-state-runtime-1.18-SNAPSHOT.jar from the opt/ folder of your Flink distribution, to the lib/ folder.
- Set the property queryable-state.enable to true.

## 09. Queryable State Stream

```
stream.keyBy(value -> value.f0).asQueryableState("query-name");
```

```
// Shortcut for explicit ValueStateDescriptor variant QueryableStateStream
asQueryableState(String queryableStateName)
// ValueState QueryableStateStream
asQueryableState( String queryableStateName, ValueStateDescriptor
stateDescriptor)
// ReducingState QueryableStateStream
asQueryableState( String queryableStateName, ReducingStateDescriptor
stateDescriptor)
```

## 09. Queryable State Stream

```
stream.keyBy(value -> value.f0).asQueryableState("query-name");
```

("apple", 1)

("banana", 2)

("apple", 3)

("banana", 4)

("apple", 5)

## 09. Managed Keyed State

```
StateDescriptor.setQueryable(String queryableStateName)
```

```
ValueStateDescriptor<Tuple2<Long, Long>> descriptor =  
    new ValueStateDescriptor<>(  
        "average", // the state name  
        TypeInformation.of(new TypeHint<Tuple2<Long, Long>>() {})); // type  
    information  
    descriptor.setQueryable("query-name"); // queryable state name
```

## 09. Querying State

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-queryable-state-client-java</artifactId>
  <version>1.18-SNAPSHOT</version>
</dependency>
```

```
QueryableStateClient client = new QueryableStateClient(tmHostname, proxyPort);
```

## 09. Querying State

```
CompletableFuture<S> getKvState(  
    JobID jobId,  
    String queryableStateName,  
    K key,  
    TypeInformation<K> keyTypeInfo,  
    StateDescriptor<S, V> stateDescriptor)
```

## 09. JobID

- Flink Web UI
  - `http://<jobmanager-ip>:8081`
- Flink REST API
- Programming
  - `StreamExecutionEnvironment env =  
StreamExecutionEnvironment.getExecutionEnvironment();`
  - `JobExecutionResult result = env.execute("My Flink Job");`
  - `JobID jobId = result.getJobID();`

## 09. REST API

- /jobs
- /jobs/{jobid}
- /jobs/{jobid}/vertices
- /jobs/{jobid}/vertices/{vertexid}
- /jobs/{jobid}/vertices/{vertexid}/metrics
- /jobs/{jobid}/exceptions

## 09. Configuration

- State Server
  - ports
  - network-threads
  - query-threads
- Proxy
  - ports
  - network-threads
  - query-threads

## 09. Limitations

- The queryable state life-cycle is bound to The life-cycle of the job
- KvState happen via a simple tell
- The server and client keep track of statistics for queries(disabled by default)

# Chapter 4.

## 10. 데이터 탑 & 직렬화

## 10. Data Types & Serialization

- Type Descriptors
- Generic Type Extraction
- Type Serialization Framework

## 10. Type Descriptors

```
StreamExecutionEnvironment env =  
StreamExecutionEnvironment.getExecutionEnvironment();  
  
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);  
DataStream<Integer> integerStream = env.fromCollection(integers,  
TypeInformation.of(Integer.class));  
  
integerStream.print();
```

## 10. Generic Type Extraction

```
DataStream<String> dataStream = env.fromElements("hello", "world");
```

```
DataStream<List<String>> dataStream = env.fromElements(  
    Arrays.asList("hello", "world"),  
    Arrays.asList("flink", "example")  
).returns(new TypeHint<List<String>>() {});
```

## 10. Type Serialization Framework

```
public class CustomTypeSerializer extends TypeSerializer<CustomType> {  
    // Implement the methods required by the TypeSerializer interface  
}
```

```
DataStream<CustomType> stream = env.addSource(new  
CustomSourceFunction()  
    .returns(TypeInformation.of(CustomType.class))  
    .with(new CustomTypeSerializer());
```

## 10. Supported Data Types

- Java Tuples and Scala Case Classes
- Java POJOs
- Primitive Types
- Regular Classes
- Values
- Hadoop Writables
- Special Types – Either(Java)

## 10. Data Types - Values

```
public class SparseVector implements Value {  
    private Map<Integer, Double> nonZeroEntries;  
    public SparseVector() {  
        this.nonZeroEntries = new HashMap<>();  
    }  
    public void set(int index, double value) {  
        if (value != 0.0) { nonZeroEntries.put(index, value); }  
    }  
    public double get(int index) {  
        return nonZeroEntries.getOrDefault(index, 0.0);  
    }
```

## 10. Data Types - Values

```
@Override public void write(DataOutput out) throws IOException {  
    out.writeInt(nonZeroEntries.size());  
    for (Map.Entry<Integer, Double> entry : nonZeroEntries.entrySet()) {  
        out.writeInt(entry.getKey()); out.writeDouble(entry.getValue());  
    } }  
  
@Override public void read(DataInput in) throws IOException {  
    int size = in.readInt();  
    nonZeroEntries = new HashMap<>(size);  
    for (int i = 0; i < size; i++) {  
        int index = in.readInt(); double value = in.readDouble();  
        nonZeroEntries.put(index, value); } } }
```

## 10. Type Erasure

- In JVM
  - `DataStream<String> == DataStream<Long>`

## 10. Type Inference

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

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);  
DataStream<Integer> dataStream = env.fromCollection(list);
```

```
TypeInformation<?> type = dataStream.getType();  
System.out.println(type); // Integer 출력
```

## 10. Type Inference

```
public class MyMapFunction implements MapFunction<Long, String>,  
ResultTypeQueryable<String> {  
    @Override  
    public String map(Long value) {  
        return "The value is: " + value;  
    }  
    @Override  
    public TypeInformation<String> getProducedType() {  
        return BasicTypeInfo.STRING_TYPE_INFO;  
    }  
}
```

## 10. Type Handling

- Registering subtypes

```
StreamExecutionEnvironment env =  
    StreamExecutionEnvironment.getExecutionEnvironment();  
env.registerType(MySubType.class);
```

- Registering custom serializers

```
env.getConfig().addDefaultKryoSerializer(MyCustomType.class,  
    MyCustomSerializer.class);
```

## 10. Type Handling

- Adding Type Hints

```
DataStream<Tuple2<String, Integer>> stream =  
env.fromElements(Tuple2.of("hello", 1))  
.returns(new TypeHint<Tuple2<String, Integer>>() {});
```

- Manually creating a TypeInformation

```
TypeInformation<Tuple2<String, Integer>> info = new TupleTypeInfo<>(  
    BasicTypeInfo.STRING_TYPE_INFO,  
    BasicTypeInfo.INT_TYPE_INFO  
>;
```

## 10. TypeInformation class

- Basic types: All Java primitives and their boxed form, plus void, String, Date, BigDecimal, and BigInteger
- Primitive arrays and Object arrays
- Composite types
  - Flink Java Tuples: max 25 fields, null fields not supported
  - Row: with arbitrary number of fields, support for null fields
  - POJOs: classes that follow a certain bean-like pattern
- Auxiliary types (Option, Either, Lists, Maps, ⋯)
- Generic types: serialized by Kryo

## 10. Creating a TypeInformation or TypeSerializer

```
StreamExecutionEnvironment env =  
    StreamExecutionEnvironment.getExecutionEnvironment();  
ExecutionConfig config = env.getConfig();  
(getRuntimeContext().getExecutionConfig())  
  
// Create a TypeInformation for a non-generic type  
TypeInformation<String> stringTypeInfo = TypeInformation.of(String.class);  
// Create a TypeSerializer for the non-generic type  
TypeSerializer<String> stringSerializer = stringTypeInfo.createSerializer(config);
```

## 10. Creating a TypeInformation or TypeSerializer

```
// Create a TypeInformation for a generic type
TypeInformation<Tuple2<String, Double>> tupleTypeInfo =
    TypeInformation.of(new TypeHint<Tuple2<String, Double>>() {});
// Create a TypeSerializer for the generic type
TypeSerializer<Tuple2<String, Double>> tupleSerializer =
    tupleTypeInfo.createSerializer(config);
```

## 10. Defining Type Information using a Factory

```
TypeHierarchy typeHierarchy = ...; // Type Hierarchy를 나타내는 클래스
```

```
TypeInfoFactory factory = null; // Type Information Factory를 나타내는 클래스
```

```
// Type Hierarchy를 위로 올라가면서 가장 가까운 Factory를 찾습니다.
```

```
while (typeHierarchy != null) { TypeInfoFactory currentFactory =
```

```
typeHierarchy.getCurrentType().getFactory();
```

```
if (currentFactory != null) {
```

```
factory = currentFactory;
```

```
break;
```

```
}
```

```
typeHierarchy = typeHierarchy.getParent(); }
```

## 10. Defining Type Information using a Factory

```
// built-in factory가 있다면, 그것이 가장 높은 우선순위를 가집니다.  
TypeInfoFactory builtInFactory = ...; // built-in factory 를 얻거나 생성  
if (builtInFactory != null) {  
    factory = builtInFactory;  
}  
  
// factory는 Flink의 built-in type보다 높은 우선순위를 가집니다.
```

## 10. Defining Type Information using a Factory

```
@TypeInfo(MyTupleTypeFactory.class)
public class MyTuple<T0, T1> {
    public T0 myfield0;
    public T1 myfield1;
}
```

## 10. Defining Type Information using a Factory

```
public class MyTupleTypeInfoFactory extends TypeInfoFactory<MyTuple> {  
  
    @Override  
    public TypeInfo<MyTuple> createTypeInfo(Type t, Map<String,  
        TypeInfo<?>> genericParameters) {  
        return new MyTupleTypeInfo(genericParameters.get("T0"),  
            genericParameters.get("T1"));  
    }  
}
```

## 10. Defining Type Information using a Factory

```
public class MyPojo {  
    public int id;  
  
    @TypeInfo(MyTupleTypeInfoFactory.class)  
    public MyTuple<Integer, String> tuple;  
}
```

## 10. POJOs, Kryo & Avro

- If Kryo is not able to handle the POJO types,

PojoTypeInfo to serialize the POJO

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

- If entire POJO Type to be treated by the Kryo serializer

```
env.getConfig().enableForceKryo();
```

## 10. POJOs, Kryo & Avro

- If Kryo is not able to serialize the POJO,  
add a custom serializer to Kryo

```
env.getConfig().addDefaultKryoSerializer(Class<?> type, Class<? extends  
Serializer<?>> serializerClass);
```

Example)

```
TypeInformation<CustomObject> typeInformation =  
TypeInformation.of(CustomObject.class);  
config.registerTypeWithKryoSerializer(CustomObject.class,  
CustomObjectSerializer.class);  
env.getConfig().disableGenericTypes();
```

## 10. Supported data types for (state) schema evolution

- POJO types
  - Fields can be removed.

Once removed, the previous value for the removed field will be dropped in future checkpoints and savepoints.
  - New fields can be added. The new field will be initialized to the default value for its type, as defined by Java.
  - Declared fields types cannot change.
  - Class name of the POJO type cannot change, including the namespace of the class.

When restoring with Flink versions older than 1.8.0,  
the schema cannot be changed.

## 10. Supported data types for (state) schema evolution

- Avro types
  - Fully supports evolving schema of Avro type state
  - Limitation: When Avro generated classes used as the state type
    - When the job is restored
      - Cannot be relocated
      - Cannot have different namespaces.

# Chapter 4.

## 11. Side Outputs

## 11. Side Outputs

- Main Stream 외에도  
여러 개의 추가적인 Stream 생성 가능  
Main Stream 뿐 아니라 여러 Side Outputs 간 type은 다 다를 수 있음
- 분할(split)할 경우 주로 사용

## 11. Output Tags

```
// this needs to be an anonymous inner class,  
// so that we can analyze the type  
OutputTag<String> outputTag = new OutputTag<String>("side-output") {};
```

## 11. Possible Functions

- ProcessFunction
- KeyedProcessFunction
- CoProcessFunction
- KeyedCoProcessFunction
- ProcessWindowFunction
- ProcessAllWindowFunction

## 11. ProcessFunction Example

```
final OutputTag<String> outputTag = new OutputTag<String>("side-output"){};  
  
SingleOutputStreamOperator<Integer> mainDataStream = input  
.process(new ProcessFunction<Integer, Integer>() {  
    @Override  
    public void processElement(Integer value, Context ctx,  
        Collector<Integer> out) throws Exception {  
        out.collect(value); // emit data to regular output  
        ctx.output(outputTag, "sideout-" + String.valueOf(value)); // emit data to side  
        output }});
```

## 11. Use Side Outputs

```
final OutputTag<String> outputTag = new OutputTag<String>("side-output"){};
```

```
SingleOutputStreamOperator<Integer> mainDataStream = ...;
```

```
DataStream<String> sideOutputStream =  
mainDataStream.getSideOutput(outputTag);
```

# Chapter 5.

## 01. Pattern API

# 01. Dependency & Implement

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-cep</artifactId>
    <version>1.18-SNAPSHOT</version>
</dependency>
```

- Proper `equals()` and `hashCode()`

## 01. Example

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

Pattern<Event, ?> pattern = Pattern.<Event>begin("start")
    .where(SimpleCondition.of(event -> event.getId() == 42))
    .next("middle")
    .subtype(SubEvent.class)
    .where(SimpleCondition.of(subEvent -> subEvent.getVolume() >= 10.0))
    .followedBy("end")
    .where(SimpleCondition.of(event -> event.getName().equals("end")));

PatternStream<Event> patternStream = CEP.pattern(input, pattern);
```

## 01. Example

```
DataStream<Alert> result = patternStream.process(  
    new PatternProcessFunction<Event, Alert>() {  
        @Override  
        public void processMatch(  
            Map<String, List<Event>> pattern,  
            Context ctx,  
            Collector<Alert> out) throws Exception {  
            out.collect(createAlertFrom(pattern));  
        }  
    });
```

# 01. The Pattern API

- Patterns & Pattern Sequence
- Match
- Individual Patterns & Complex Patterns

# 01. Individual Patterns

- Pattern
  - Singleton
  - Looping
  - For example, "a b+ c? d"
    - Singleton: a, c?, d
    - Looping: b+
- Quantifiers
  - pattern.oneOrMore()
  - pattern.times(#ofTimes)
  - pattern.times(#fromTimes, #toTimes)

## 01. Individual Patterns

- Greedy and Optional
  - pattern.greedy()
  - pattern.optional()

## 01. Quantifiers

```
// expecting 4 occurrences
start.times(4);

// expecting 0 or 4 occurrences
start.times(4).optional();

// expecting 2, 3 or 4 occurrences and repeating as many as possible
start.times(2, 4).greedy();

// expecting 2, 3 or 4 occurrences
start.times(2, 4);

// expecting 0, 2, 3 or 4 occurrences
start.times(2, 4).optional();
```

# 01. Conditions

- Methods
  - where()
  - or()
  - until()
- Types
  - IterativeConditions
  - SimpleConditions

# 01. Iterative Conditions

```
middle.oneOrMore()  
.subtype(SubEvent.class)  
.where(new IterativeCondition<SubEvent>() {  
    @Override  
    public boolean filter(SubEvent value, Context<SubEvent> ctx) throws  
    Exception {  
        if (!value.getName().startsWith("foo")) {  
            return false;  
        }  
    }  
}
```

## 01. Iterative Conditions

```
double sum = value.getPrice();
for (Event event : ctx.getEventsForPattern("middle")) {
    sum += event.getPrice();
}
return Double.compare(sum, 5.0) < 0;
});
```

## 01. Iterative Conditions

```
middle.oneOrMore()  
    .subtype(SubEvent.class)  
    .where(new IterativeCondition<SubEvent>() {  
        @Override  
        public boolean filter(SubEvent value, Context<SubEvent> ctx) throws  
        Exception {  
            if (!value.getName().startsWith("foo")) {  
                return false;  
            }  
        }  
    })
```

## 01. Simple Conditions

pattern =

```
start.where(SimpleCondition.of(value -> value.getName().startsWith("foo")));
```

- Combining Conditions

```
pattern.where(SimpleCondition.of(value -> ... /*some condition*/))  
.or(SimpleCondition.of(value -> ... /*some condition*/));
```

```
pattern.where(SimpleCondition.of(value -> ... /*some condition 1*/))  
.or(SimpleCondition.of(value -> ... /*some condition 2*/))  
.where(SimpleCondition.of(value -> ... /*some condition 3*/));
```

# 01. Stop Conditions

- Looping Patterns
  - oneOrMore()
  - oneOrMore().optional()
- Example
  - Pattern
    - "(a+ until b)" (하나 이상의 "a"가 "b"가 나올 때까지)
  - Event Sequence
    - "a1" "c" "a2" "b" "a3"
  - Result
    - {a1 a2} {a1} {a2} {a3}

## 01. Methods in Individual Patterns

- where(condition): 현재 패턴에 대한 조건을 정의
- or(condition): 기존 조건과 OR 연산되는 새로운 조건을 추가
- until(condition): 반복 패턴에 대한 중지 조건
- subtype(subClass): 현재 패턴에 대한 하위 유형(type) 조건
- oneOrMore(), timesOrMore(#times), times(#ofTimes),  
times(#fromTimes, #toTimes)
- optional()
- greedy()

## 01. Methods in Combining Patterns

- `next()`, `followedBy()`, `followedByAny()`
  - `next()`는 엄격한 연속성[Strict Contiguity]
  - `followedBy()`는 느슨한 연속성[Relaxed Contiguity]
  - `followedByAny()`는 비결정적 느슨한 연속성  
[Non-Deterministic Relaxed Contiguity]
- `notNext()`, `notFollowedBy()`
- `within()`: 패턴이 유효한 시간 제약을 정의

## 01. followedBy() vs followedByAny()

- Example1
  - Event Sequence: "a", "c", "b1", "b2"
  - followedBy(): {a b1}
  - followedByAny(): {a b1}, {a b2}
- Example2
  - Event Sequence: "a1", "c", "b1", "b2", "a2", "b3"
  - followedBy(): {a1 b1}, {a2 b3}
  - followedByAny(): {a1 b1}, {a1 b2}, {a2 b3}

## 01. Contiguity within looping patterns

- Pattern: a b+ c
- Event Sequence: "a", "b1", "d1", "b2", "d2", "b3", "c"
- Result
  - Strict Contiguity: {a b1 c}, {a b2 c}, {a b3 c}
  - Relaxed Contiguity: {a b1 c}, {a b1 b2 c}, {a b1 b2 b3 c}, {a b2 c}, {a b2 b3 c}, {a b3 c}
  - Non-Deterministic Relaxed Contiguity: {a b1 c}, {a b1 b2 c}, {a b1 b3 c}, {a b1 b2 b3 c}, {a b2 c}, {a b2 b3 c}, {a b3 c}

## 01. consecutive()

C D A1 A2 A3 D A4 B -> {C A1 B}, {C A1 A2 B}, {C A1 A2 A3 B}

```
Pattern.<Event>begin("start")
    .where(SimpleCondition.of(value -> value.getName().equals("c")))
    .followedBy("middle")
    .where(SimpleCondition.of(value -> value.getName().equals("a")))
    .oneOrMore()
    .consecutive() // 엄격한 연속성을 적용
    .followedBy("end1")
    .where(SimpleCondition.of(value -> value.getName().equals("b")));
```

## 01. consecutive()

C D A1 A2 A3 D A4 B -> {C A1 B}, {C A1 A2 B}, {C A1 A2 A3 B}, {C A1 A2 A3 A4 B}

```
Pattern.<Event>begin("start")
    .where(SimpleCondition.of(value -> value.getName().equals("c")))
    .followedBy("middle")
    .where(SimpleCondition.of(value -> value.getName().equals("a")))
    .oneOrMore()
    .consecutive() // 연속성을 지정하지 않았으므로 느슨한 연속성이 적용
    .followedBy("end1")
    .where(SimpleCondition.of(value -> value.getName().equals("b")));
```

## 01. allowCombinations()

C D A1 A2 A3 D A4 B -> {C A1 B}, {C A1 A2 B}, {C A1 A3 B}, {C A1 A4 B}, {C A1 A2 A3 B}, {C A1 A2 A4 B}, {C A1 A3 A4 B}, {C A1 A2 A3 A4 B}

```
Pattern.<Event>begin("start")
    .where(SimpleCondition.of(value -> value.getName().equals("c")))
    .followedBy("middle")
    .where(SimpleCondition.of(value -> value.getName().equals("a")))
    .oneOrMore()
    .allowCombinations() // 비결정적 느슨한 연속성을 적용
    .followedBy("end1")
    .where(SimpleCondition.of(value -> value.getName().equals("b")));
```

## 01. allowCombinations()

C D A1 A2 A3 D A4 B -> {C A1 B}, {C A1 A2 B}, {C A1 A2 A3 B}, {C A1 A2 A3 A4 B}

```
Pattern.<Event>begin("start")
    .where(SimpleCondition.of(value -> value.getName().equals("c")))
    .followedBy("middle")
    .where(SimpleCondition.of(value -> value.getName().equals("a")))
    .oneOrMore()
    .allowCombinations() // 연속성을 지정하지 않았으므로 느슨한 연속성이 적용
    .followedBy("end1")
    .where(SimpleCondition.of(value -> value.getName().equals("b")));
```

## 01. optional()

C D A1 A2 A3 D A4 B -> {C A1 B}, {C A1 A2 B}, {C A1 A2 A3 B}, {C A1 A2 A3 A4 B},  
{C B}

```
Pattern.<Event>begin("start")
    .where(SimpleCondition.of(value -> value.getName().equals("c")))
    .followedBy("middle")
    .where(SimpleCondition.of(value -> value.getName().equals("a")))
    .oneOrMore()
    .optional()
    .followedBy("end1")
    .where(SimpleCondition.of(value -> value.getName().equals("b")));
```

## 01. optional().allowCombinations()

C D A1 A2 A3 D A4 B -> {C A1 B}, {C A1 A2 B}, {C A1 A2 A3 B}, {C A1 A2 A3 A4 B},  
+ Alpha

```
Pattern.<Event>begin("start")
    .where(SimpleCondition.of(value -> value.getName().equals("c")))
    .followedBy("middle")
    .where(SimpleCondition.of(value -> value.getName().equals("a")))
    .oneOrMore()
    .optional().allowCombinations()
    .followedBy("end1")
    .where(SimpleCondition.of(value -> value.getName().equals("b")));
```

## 01. optional().allowCombinations()

C D A1 A2 A3 D A4 B -> {C A1 B}, {C A2 B}, {C A3 B}, {C A4 B},  
{C A1 A2 B}, {C A1 A3 B}, {C A1 A4 B}, {C A2 A3 B}, {C A2 A4 B}, {C A3 A4 B},  
{C A1 A2 A3 B}, {C A1 A2 A4 B}, {C A1 A3 A4 B}, {C A2 A3 A4 B},  
{C A1 A2 A3 A4 B}

## 01. Group of patterns

- begin(#name) / begin(#pattern\_sequence)
- next(#name) / next(#pattern\_sequence)
- followedBy(#name) / followedBy(#pattern\_sequence)
- followedByAny(#name) / followedByAny(#pattern\_sequence)
- notNext() / notFollowedBy()
- within(time)

## 01. After Match Skip Strategy

- Pattern: b+ c
- Event Stream: b1 b2 b3 c
- Strategies
  - NO\_SKIP: "b1 b2 b3 c", "b2 b3 c", "b3 c"
  - SKIP\_TO\_NEXT: "b1 b2 b3 c", "b2 b3 c", "b3 c"
  - SKIP\_PAST\_LAST\_EVENT: "b1 b2 b3 c"
  - SKIP\_TO\_FIRST[b]: "b1 b2 b3 c", "b2 b3 c", "b3 c"
  - SKIP\_TO\_LAST[b]: "b1 b2 b3 c", "b3 c"

## 01. After Match Skip Strategy

- Pattern: a b+
- Event Stream: a b1 b2 b3
- Strategies
  - NO\_SKIP: "a b1", "a b1 b2", "a b1 b2 b3"
  - SKIP\_TO\_NEXT: "a b1"

## 01. After Match Skip Strategy

- Pattern:  $(a \mid b \mid c) (b \mid c)^*$ .greedy
- Event Stream: a b c1 c2 c3 d
- Strategies
  - NO\_SKIP: "a b c1 c2 c3 d", "b c1 c2 c3 d", "c1 c2 c3 d"
  - SKIP\_TO\_FIRST[c\*]: "a b c1 c2 c3 d", "c1 c2 c3 d"

## 01. After Match Skip Strategy

```
AfterMatchSkipStrategy skipStrategy =  
    AfterMatchSkipStrategy.skipToFirst(patternName).throwExceptionOnMiss();  
  
Pattern.begin("patternName", skipStrategy);
```

## 01. Detecting Patterns

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

```
Pattern<Event, ?> pattern = ...;
```

```
EventComparator<Event> comparator = ...; // optional
```

```
PatternStream<Event> patternStream = CEP.pattern(input, pattern, comparator);
```

# 01. Detecting Patterns – Selecting from Patterns

- PatternProcessFunction
  - Map<String, List<IN>>
- Context

## 01. Detecting Patterns – Selecting from Patterns

```
class MyPatternProcessFunction<IN, OUT> extends PatternProcessFunction<IN,  
OUT> {  
    @Override  
    public void processMatch(Map<String, List<IN>> match, Context ctx,  
    Collector<OUT> out) throws Exception {  
        IN startEvent = match.get("start").get(0);  
        IN endEvent = match.get("end").get(0);  
        out.collect(OUT(startEvent, endEvent));  
    }  
}
```

## 01. Detecting Patterns – Selecting from Patterns

```
PatternStream<Event> patternStream = CEP.pattern(input, pattern);
DataStream<ComplexEvent> resultStream =
    patternStream.process(new MyPatternProcessFunction());
```

## 01. Detecting Patterns – Handling Timed Out Partial Patterns

```
class MyPatternProcessFunction<IN, OUT> extends PatternProcessFunction<IN,  
OUT> implements TimedOutPartialMatchHandler<IN> {  
    ...  
    @Override  
    public void processTimedOutMatch(Map<String, List<IN>> match, Context ctx)  
throws Exception {  
    // 타임아웃된 매치 처리 로직  
    IN startEvent = match.get("start").get(0);  
    ctx.output(outputTag, T(startEvent));  
}  
}
```

## 01. Detecting Patterns – Handling Timed Out Partial Patterns

```
PatternStream<Event> patternStream = CEP.pattern(input, pattern);
SingleOutputStreamOperator<ComplexEvent> flatResult =
    patternStream.flatSelect( ... )
```

- flatSelect()
  - PatternFlatTimeoutFunction {}
    - timeout()
  - PatternFlatSelectFunction{}
    - flatSelect()

# Chapter 5.

## 04. Time in CEP

## 04. Handling Lateness in Event Time

```
PatternStream<Event> patternStream = CEP.pattern(input, pattern);
```

```
OutputTag<String> lateDataOutputTag = new OutputTag<String>("late-data"){};
```

```
SingleOutputStreamOperator<ComplexEvent> result = patternStream  
    .sideOutputLateData(lateDataOutputTag)  
    .select(  
        new PatternSelectFunction<Event, ComplexEvent>() {...}  
    );
```

```
DataStream<String> lateData = result.getSideOutput(lateDataOutputTag);
```

## 04. Time context

```
@PublicEvolving  
public interface TimeContext {  
    long timestamp();  
    long currentProcessingTime();  
}
```

## 04. Optional Configuration

- cache capacity of SharedBuffer
  - accelerate the CEP operate process speed
  - limit the number of elements of cache in pure memory
- state.backend.type: rocksdb
  - only effective

## 04. Optional Configuration - SharedBuffer

- Slots
  - entry-slot (entryCache)
  - event-slot (eventsBufferCache)
- Configs
  - pipeline.cep.sharedbuffer.cache.entry-slots
  - pipeline.cep.sharedbuffer.cache.event-slots
  - pipeline.cep.sharedbuffer.cache.statistics-interval

# Chapter 6.

## 01. Overview

## 01. Table API

- Flink ML's API is based on Flink's Table API
- Table API is a language-integrated query API for JAVA, Scala and Python
  - selection, filter, and join in a very intuitive way
- Table API(data types) + Flink ML(Vector)
  - CHAR, VARCHAR, STRING, BOOLEAN, BYTES, DECIMAL, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DATE, TIME, TIMESTAMP, TIMESTAMP\_LTZ, INTERVAL, ARRAY, MULTISET, MAP, ROW, RAW, structured types
  - Vector
- The Table API integrates seamlessly with Flink's DataStream API

## 01. Stage

- Stage(Interface)
  - Pipeline 또는 Graph의 노드
  - Flink ML의 기본 구성 요소
  - Stage 자체는 개념적인(concept) 인터페이스로, 실제 기능은 없음
  - Subclasses
    - Estimator: fit()
    - AlgoOperator: transform()
    - Transformer
    - Model: getModelData(), setModelData()

## 01. Stage

- multi-input multi-output
  - vs a record in the output typically corresponds to one record in the input
- AlgoOperator vs Transformer vs Model

## 01. Stage - Example

```
// Suppose SumModel is a concrete subclass of Model,  
// SumEstimator is a concrete subclass of Estimator.
```

```
Table trainData = ...;
```

```
Table predictData = ...;
```

```
SumEstimator estimator = new SumEstimator();
```

```
SumModel model = estimator.fit(trainData);
```

```
Table predictResult = model.transform(predictData)[0];
```

# 01. Builders

- Flink ML's APIs
  - Help to manage the relationship and structure of stages
  - Pipeline and Graph

## 01. Stage class

```
@PublicEvolving
public interface Stage<T extends Stage<T>> extends WithParams<T>, Serializable {
    /** Saves the metadata and bounded data of this stage to the given path. */
    void save(String path) throws IOException;
}
```

## 01. Param

- Param
  - the definition of a parameter
  - name
  - class
  - description
  - default value
  - the validator

## 01. Parameter of algorithm

- Set the parameter of an algorithm
  - Invoke the parameter's specific set method.
    - For example, users can directly invoke `setK()` method on that KMeans instance.
    - Pass a parameter map containing new values to the stage through `ParamUtils.updateExistingParams()` method
  - The Model would inherit the Estimator object's parameters.

# Chapter 6.

## 02. Pipeline & Graph

## 02. Pipeline

- An ordered list of stages
  - Estimator, Model, Transformer, AlgoOperator

## 02. Pipeline - Example

```
// Suppose SumModel is a concrete subclass of Model,  
// SumEstimator is a concrete subclass of Estimator.
```

```
Model modelA = new SumModel().setModelData(tEnv.fromValues(10));
```

```
Estimator estimatorA = new SumEstimator();
```

```
Model modelB = new SumModel().setModelData(tEnv.fromValues(30));
```

```
List<Stage<?>> stages = Arrays.asList(modelA, estimatorA, modelB);
```

```
Estimator<?, ?> estimator = new Pipeline(stages);
```

## 02. Pipeline - Example

// 데이터 로딩 및 초기화

```
Table trainData = ...; // 학습 데이터
```

```
Table predictData = ...; // 예측을 위한 데이터
```

...

// Pipeline을 사용하여 모델 학습

```
Model model = estimator.fit(trainData);
```

// 학습된 모델을 사용하여 데이터 변환

```
Table predictResult = model.transform(predictData);
```

## 02. Pipeline - Diagram



## 02. Graph

- A DAG of stages
  - Estimator, Model, Transformer, AlgoOperator
- The stages are executed in a topologically-sorted order

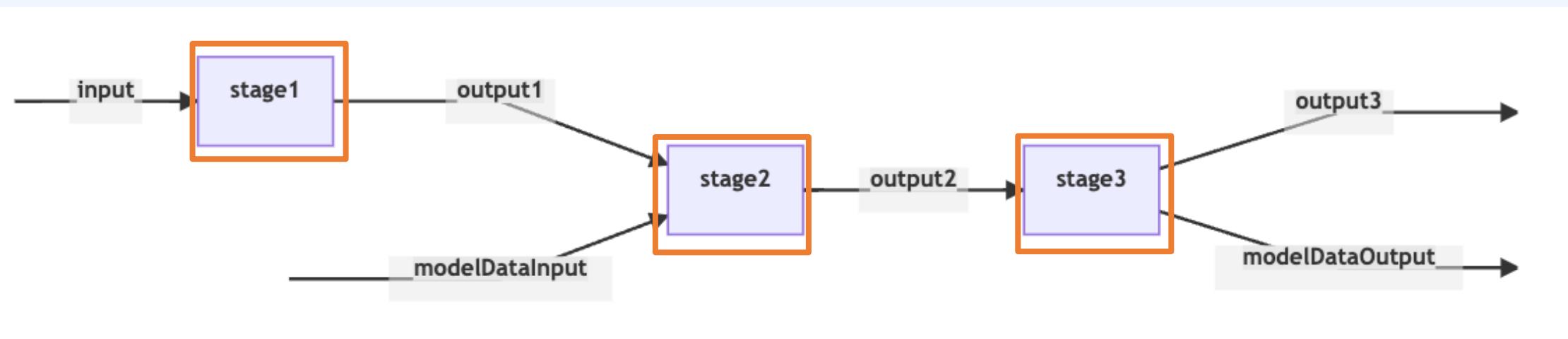
## 02. Graph - Example

```
// Suppose SumModel is a concrete subclass of Model.
```

```
GraphBuilder builder = new GraphBuilder();
// Creates nodes.

SumModel stage1 = new SumModel().setModelData(tEnv.fromValues(1));
SumModel stage2 = new SumModel();
SumModel stage3 = new SumModel().setModelData(tEnv.fromValues(3));
```

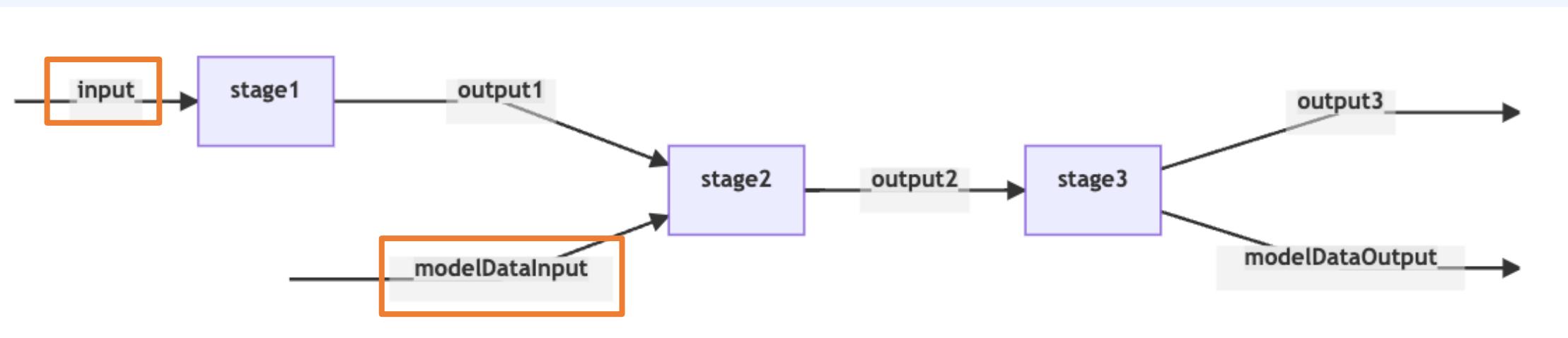
## 02. Graph - Diagram



## 02. Graph - Example

```
// Creates inputs and modelDataInputs.  
TableId input = builder.createTableId();  
TableId modelDataInput = builder.createTableId();
```

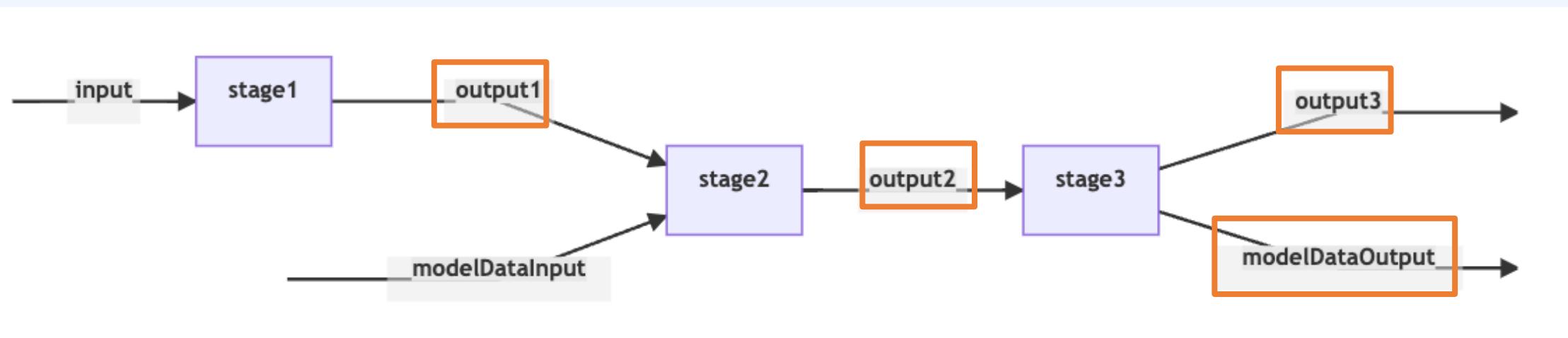
## 02. Graph - Diagram



## 02. Graph - Example

```
// Feeds inputs and gets outputs.  
TableId output1 = builder.addAlgoOperator(stage1, input)[0];  
TableId output2 = builder.addAlgoOperator(stage2, output1)[0];  
builder.setModelDataOnModel(stage2, modelDataInput);  
TableId output3 = builder.addAlgoOperator(stage3, output2)[0];  
TableId modelDataOutput = builder.getModelDataFromModel(stage3)[0];
```

## 02. Graph - Diagram



## 02. Graph - Example

```
// Builds a Model from the graph.  
TableId[] inputs = new TableId[] {input};  
TableId[] outputs = new TableId[] {output3};  
TableId[] modelDataInputs = new TableId[] {modelDataInput};  
TableId[] modelDataOutputs = new TableId[] {modelDataOutput};  
Model<?> model = builder.buildModel(inputs, outputs, modelDataInputs,  
modelDataOutputs);
```

## 02. Pipeline vs Graph

- Pipeline: Simple, Sequential
- Graph: Complex, Diverse

## 02. Pipeline vs Graph

```
pipeline = RecipePipeline()  
pipeline.add_step(BoilWater())  
pipeline.add_step(CookPasta())  
pipeline.add_step(AddSauce())  
pipeline.execute()
```

## 02. Pipeline vs Graph

```
graph = KitchenGraph()  
graph.add_task("dough", MakeDough())  
graph.add_task("bread", BakeBread(), depends_on=["dough"])  
graph.add_task("salad", MakeSalad()) graph.add_task("sandwich",  
MakeSandwich(), depends_on=["bread", "salad"])  
graph.execute()
```

# Chapter 6.

## 03. Vector

## 03. Vector

- DenseVector
- SparseVector

## 03. Vector- Example

```
int n = 4; // 벡터의 차원
```

```
int[] indices = new int[] {0, 2, 3}; // 0이 아닌 값들의 위치
```

```
double[] values = new double[] {0.1, 0.3, 0.4}; // 0이 아닌 값들
```

```
SparseVector vector = Vectors.sparse(n, indices, values); // SparseVector 생성
```

## 03. Built-in table functions

- vectorToArray
- arrayToVector

## 03. vectorToArray - Example

```
// Generates input vector data.  
List<Vector> vectors = Arrays.asList(  
    Vectors.dense(0.0, 0.0),  
    Vectors.sparse(2, new int[] {1}, new double[] {1.0}));  
  
Table inputTable = tEnv.fromDataStream(env.fromCollection(  
    vectors, VectorTypeInfo.INSTANCE)) .as("vector");  
  
  
// Converts each vector to a double array.  
Table outputTable = inputTable.select(  
    $"vector"), vectorToArray(($"vector").as("array"));
```

## 03. arrayToVector - Example

```
// Generates input double array data.  
List<double[]> doubleArrays = Arrays.asList(  
    new double[] {0.0, 0.0},  
    new double[] {0.0, 1.0});  
  
Table inputTable = tEnv.fromDataStream(  
    env.fromCollection(doubleArrays)).as("array");  
  
  
// Converts each double array to a dense vector.  
Table outputTable = inputTable.select(  
    $"array"), arrayToVector($"array").as("vector"));
```

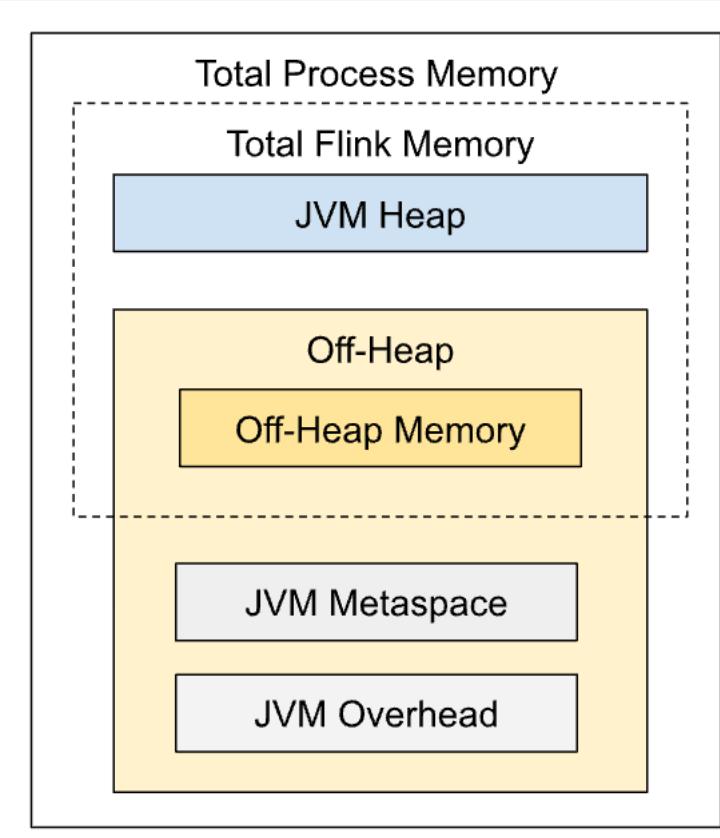
# Chapter 7.

## 01. Memory Configuration

# 01. Process Memory

- Total Process Memory
  - Total Flink Memory
    - JVM Heap
    - Off-Heap(Direct or Native)
  - JVM to run the process

# 01. Process Memory - Diagram



## 01. Configure Total Memory - Simple

for TaskManager:

[taskmanager.memory.flink.size](#)

[taskmanager.memory.process.size](#)

for JobManager:

[jobmanager.memory.flink.size](#)

[jobmanager.memory.process.size](#)

# 01. Configure Total Memory

for TaskManager:

[taskmanager.memory.flink.size](#)

[taskmanager.memory.process.size](#)

[taskmanager.memory.task.heap.size](#)  
and [taskmanager.memory.managed.size](#)

for JobManager:

[jobmanager.memory.flink.size](#)

[jobmanager.memory.process.size](#)

[jobmanager.memory.heap.size](#)

## 01. JVM Parameters

- ``-Xmx``, ``-Xms``: Framework + Task Heap Memory
- ``-XX:MaxDirectMemorySize``:  
Framework + Task Off-heap + Network Memory
- ``-XX:MaxMetaspaceSize``: JVM Metaspace

## 01. Heap vs Off-heap in Flink

- Heap
  - Object
  - Data Structure
  - State
    - (RocksDBStateBackend를 사용하지 않는 경우)
- Off-heap
  - Network Buffer
  - RocksDBStateBackend

## 01. Capped Fractionated Components

- JVM Overhead: a fraction of the total process memory
- Network memory: a fraction of the total Flink memory (only for TaskManager)
- Example 1)
  - total Process memory = 1000MB,
  - JVM Overhead min = 64MB,
  - JVM Overhead max = 128MB,
  - JVM Overhead fraction = 0.1

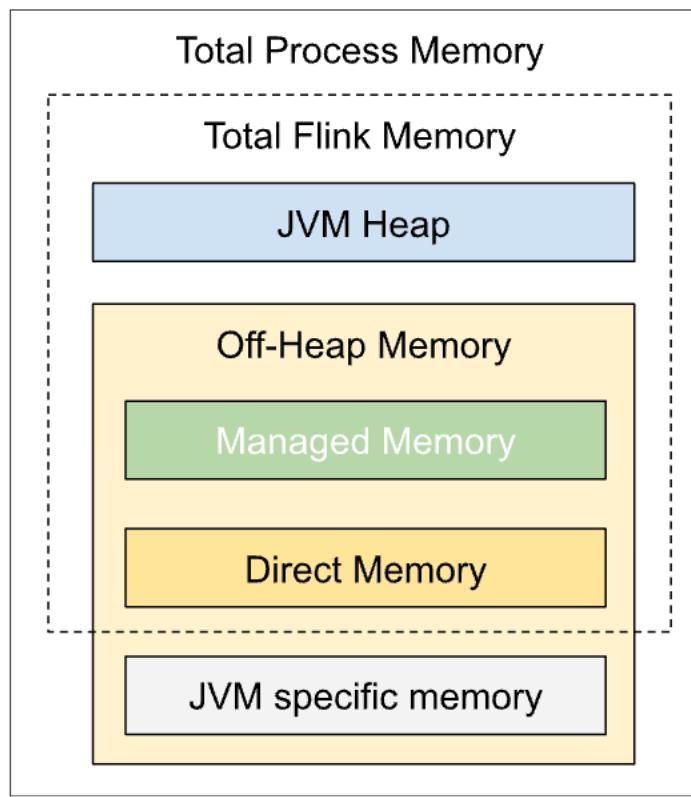
## 01. Capped Fractionated Components

- Example 2)
  - total Process memory = 1000MB,
  - JVM Overhead min = 128MB,
  - JVM Overhead max = 256MB,
  - JVM Overhead fraction = 0.1
- Example 3)
  - total Process memory = 1000MB,
  - task heap = 100MB,
  - JVM Overhead min = 64MB,
  - JVM Overhead max = 256MB,
  - JVM Overhead fraction = 0.1

# 01. JobManager vs TaskManager

- JobManager Memory
  - Scheduling
  - State
  - Fault tolerance
- TaskManager Memory
  - Execute code
  - Buffering
  - Network

# 01. TaskManager Memory - Diagram



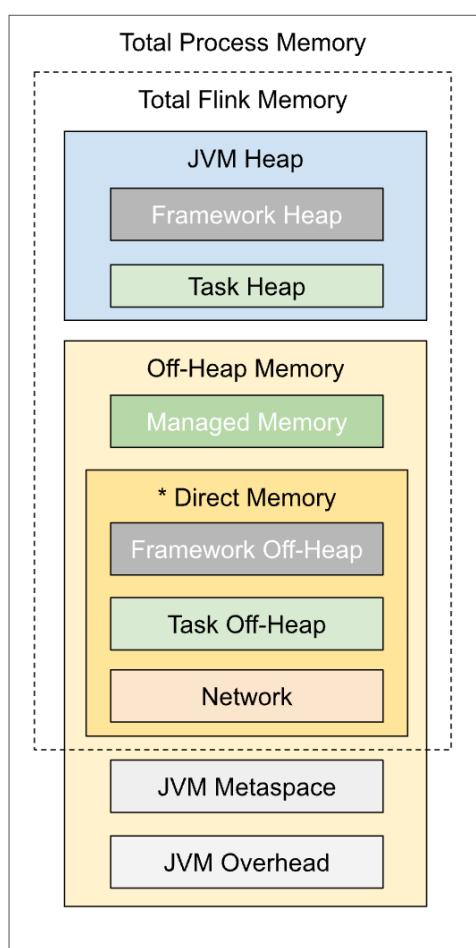
# 01. Configure Heap and Managed Memory

- Task (Operator) Heap Memory
  - taskmanager.memory.task.heap.size
- Managed Memory
  - taskmanager.memory.managed.size
  - taskmanager.memory.managed.fraction.
  - Consumer Weight
    - taskmanager.memory.managed.consumer-weights:  
STATE\_BACKEND:70, PYTHON:30
- If set above, don't set below
  - taskmanager.memory.process.size
  - taskmanager.memory.flink.size

# 01. Configure Off-heap Memory

- Total Memory
  - Direct Memory
    - Task Off-heap Memory
      - `taskmanager.memory.task.off-heap.size`
    - Framework Off-heap Memory
    - Network Memory

# 01. TaskManager Detailed Memory - Diagram



\* Notice, that the **native non-direct usage** of memory in user code can be also accounted for as a part of the **task off-heap memory**

## 01. Framework Memory

- Should not change  
the framework heap memory and framework off-heap memory
- Flink needs more memory for some internal data structures or operations.
- Flink neither isolates heap nor off-heap versions of framework and task  
memory at the moment.

## 01. Local Execution

- Task Heap Memory
- Task Off-heap Memory
- Managed Memory
- Network Memory
  
- [https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/memory/mem\\_setup\\_tm/#detailed-memory-model](https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/memory/mem_setup_tm/#detailed-memory-model)
- [https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/memory/mem\\_setup\\_tm/#local-execution](https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/memory/mem_setup_tm/#local-execution)

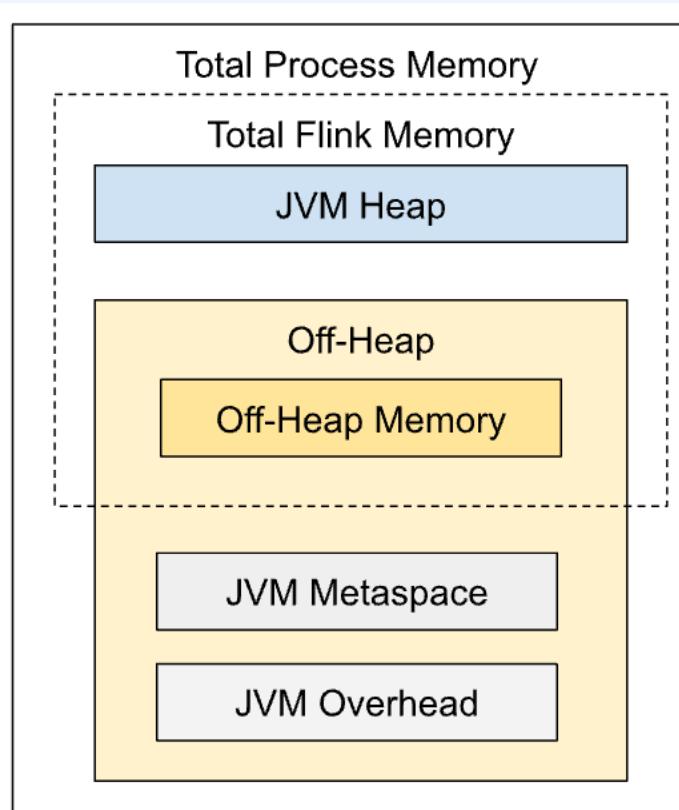
# 01. JobManager Memory

- Resource Manager
- Dispatcher
- JobMaster

## 01. Configure Total Memory

- JobManager process using local execution mode  
you do not need to configure memory options, they will **have no effect**.

# 01. JobManager Detailed Memory - Diagram



## 01. Configure JVM Heap

- Another way to set up the memory for the JobManager is to specify explicitly the JVM Heap size
  - `jobmanager.memory.heap.size`
- Control over the available JVM Heap
  - Flink framework
  - User code executed during job submission or in checkpoint completion callbacks
- JVM Heap size via the JVM parameters `-Xms` and `-Xmx`

## 01. Configure Off-heap Memory

- Off-heap memory: jobmanager.memory.off-heap.size
    - JVM direct memory:  
jobmanager.memory.enable-jvm-direct-memory-limit
    - Native memory
  - Off-heap memory consumption:
    - Flink framework dependencies
    - User code executed during job submission  
or in checkpoint completion callbacks
- [https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/memory/mem\\_setup\\_jobmanager/#detailed-configuration](https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/memory/mem_setup_jobmanager/#detailed-configuration)

## 01. Memory tuning guide

- Configure memory for standalone deployment
  - taskmanager.memory.flink.size
  - jobmanager.memory.flink.size
  - JVM metaspace
- Configure memory for containers (Kubernetes or Yarn)
  - taskmanager.memory.process.size
  - jobmanager.memory.process.size

# 01. Memory tuning guide

- Configure memory for state backends
  - HashMap state backend
    - Managed memory to zero
  - RocksDB state backend
    - state.backend.rocksdb.memory.managed

# 01. Troubleshooting

- IllegalStateException
- OutOfMemoryError: Java heap space
  - Total memory
  - Task heap memory for TaskManagers
  - JVM heap memory for JobManagers
- OutOfMemoryError: Direct buffer memory
  - Direct off-heap memory
- OutOfMemoryError: Metaspace
  - JVM metaspace for TaskManagers
  - JVM metaspace for JobManagers

## 01. Troubleshooting

- IOException: Insufficient number of network buffers
  - taskmanager.memory.network.min
  - taskmanager.memory.network.max
  - taskmanager.memory.network.fraction.
- Container Memory Exceeded
  - jobmanager.memory.enable-jvm-direct-memory-limit
  - MALLOC\_arena\_MAX=1

# 01. Troubleshooting - Monitoring

- JVM Heap Memory
- JVM Direct Memory
- JVM Metaspace
- Network Memory
- Container Memory

## 01. Network memory tuning guide

- Each record is sent to the next subtask compounded with other records in a network buffer
- To maintain consistent high throughput, Flink uses network buffer queues (also known as in-flight data)
- Each subtask has an input queue waiting to consume data and an output queue waiting to send data to the next subtask
- Checkpoints in Flink can only finish once all the subtasks receive all of the injected checkpoint barriers.

# 01. The Buffer Debloating Mechanism

- Added in Flink 1.14
- Automatically adjusting the amount of in-flight data
  - Calculates the maximum possible throughput for the subtask
  - Adjusts the amount of in-flight data
- Enable
  - `taskmanager.network.memory.buffer-debloat.enabled`
- Target time
  - `taskmanager.network.memory.buffer-debloat.target`

# 01. The Buffer Debloating Mechanism

- Fail
  - There will not be enough buffered data to provide full throughput.
  - There will be too many buffered in-flight data which will negatively affect the aligned checkpoint barriers propagation time or the unaligned checkpoint size.
- Varying load
  - taskmanager.network.memory.buffer-debloat.period
  - taskmanager.network.memory.buffer-debloat.samples
  - taskmanager.network.memory.buffer-debloat.threshold-percentages

# 01. The Buffer Debloating Mechanism

- Monitoring
  - estimatedTimeToConsumeBuffersMs
  - debloatedBufferSize

## 01. The Buffer Debloating Mechanism - Limitations

- Multiple inputs and unions
- Buffer size and number of buffers
- High parallelism
  - `taskmanager.network.memory.floating-buffers-per-gate`

## 01. Network buffer lifecycle

- The target size of each buffer pool
  - $\#channels * \text{taskmanager.network.memory.buffers-per-channel} + \text{taskmanager.network.memory.floating-buffers-per-gate}$

## 01. Network buffer lifecycle

- Input network buffers
  - taskmanager.network.memory.read-buffer.required-per-gate.max
- Output network buffers
  - taskmanager.network.memory.max-buffers-per-channel
  - Overdraft buffers
    - taskmanager.network.memory.max-overdraft-buffers-per-gate

## 01. The number of in-flight buffers

- Selecting the buffer size
  - If the buffer size is too small
    - flushed too frequently (execution.buffer-timeout)
    - decreased throughput
  - If the buffer size is too large
    - high memory usage
    - huge checkpoint data (for unaligned checkpoints)
    - long checkpoint time (for aligned checkpoints)
    - inefficient use of allocated memory with a small execution.buffer-timeout

## 01. The number of in-flight buffers

- Selecting the buffer count
  - taskmanager.network.memory.buffers-per-channel
  - taskmanager.network.memory.floating-buffers-per-gate
- Calculate number of buffers
  - $$\text{number\_of\_buffers} = \text{expected\_throughput} * \text{buffer\_roundtrip} / \text{buffer\_size}$$
$$\text{number\_of\_buffers} = 320\text{MB/s} * 1\text{ms} / 32\text{KB} = 10$$

## 01. Network memory tuning guide - Summary

- enabling the buffer debloating mechanism
- using the default values for max throughput
- reducing the memory segment size and/or number of exclusive buffers

# Chapter 7.

## 02. Elastic Scaling

## 02. Elastic Scaling

- Automatically adjusts the parallelism

## 02. Reactive Mode

- MVP(Minimum Viable Product) feature
- Configures a job so that it always uses all resources available in the cluster
- Restarts a job on a rescaling event,  
restoring it from the latest completed checkpoint
- Implement a powerful autoscaling mechanism
  - By having an external service monitor certain metrics
  - Replica factor of a Kubernetes deployment
  - Autoscaling group on AWS

## 02. Reactive Mode - Example

```
# Put Job into lib/ directory  
cp ./examples/streaming/TopSpeedWindowing.jar lib/  
# Submit Job in Reactive Mode  
.bin/standalone-job.sh start  
  -Dscheduler-mode=reactive  
  -Dexecution.checkpointing.interval="10s"  
  -j org.apache.flink.streaming.examples.windowing.TopSpeedWindowing  
# Start first TaskManager  
.bin/taskmanager.sh start
```

## 02. Reactive Mode - Example

```
# Start additional TaskManager  
./bin/taskmanager.sh start  
  
# Remove a TaskManager  
./bin/taskmanager.sh stop
```

## 02. Reactive Mode - Usage

- scheduler-mode to reactive
  - parallelism of individual operators in a job will be determined by the scheduler
    - The only way of influencing the parallelism is by setting a max parallelism for an operator:  $2^{15}$
- jobmanager.adaptive-scheduler.resource-wait-timeout: -1
- jobmanager.adaptive-scheduler.resource-stabilization-timeout: 0
- jobmanager.adaptive-scheduler.min-parallelism-increase: 1

## 02. Reactive Mode - Usage

- Note that such a high max parallelism might affect performance of the job, since more internal structures are needed to maintain some internal structures of Flink.

## 02. Reactive Mode - Recommendation

- Configure periodic checkpointing for stateful jobs
- Downscaling in Reactive Mode might take longer  
if the TaskManager is not properly shutdown  
(i.e., if a SIGKILL signal is used instead of a SIGTERM signal)

## 02. Reactive Mode - Limitations

- Deployment is only supported as a standalone application deployment

## 02. Adaptive Scheduler

- Adjust the parallelism of a job based on available slots
- In Reactive Mode, the configured parallelism is ignored and treated as if it was set to infinity
  - Adaptive Scheduler without Reactive Mode
    - > Using Adaptive Scheduler on a session cluster, no guarantees regarding the distribution of slots between multiple running jobs in the same session
  - Benefit of the Adaptive, it can handle TaskManager losses gracefully

## 02. Adaptive Scheduler - Caution

- Using Adaptive Scheduler directly (not through Reactive Mode) is only advised for advanced users because slot allocation on a session cluster with multiple jobs is not defined.
  - `jobmanager.adaptive-scheduler.min-parallelism-increase`

## 02. Adaptive Scheduler - Usage

- jobmanager.scheduler: adaptive
- cluster.declarative-resource-management.enabled
- [https://nightlies.apache.org/flink/flink-docs-](https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/config/#advanced-scheduling-options)  
[master/docs/deployment/config/#advanced-scheduling-options](https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/config/#advanced-scheduling-options)

## 02. Adaptive Scheduler -Limitations

- Streaming jobs only
- No support for local recovery
- No support for partial failover
- Scaling events trigger job and task restarts

# Chapter 7.

## 03. Fine-Grained Resource Management

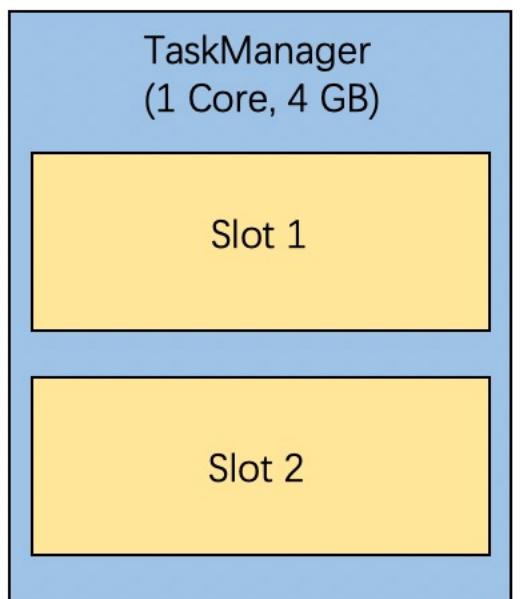
## 03. Fine-Grained Resource Management

- Fine-tune resource consumption
- MVP(Minimum Viable Product)
  - Only available to DataStream API

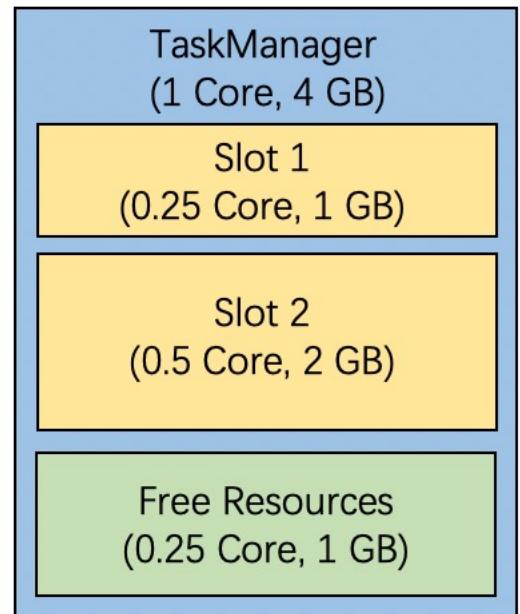
## 03. Application Scenarios

- Tasks have significantly different parallelisms.
- The resource needed for an entire pipeline is too much to fit into a single slot/task manager.
- Batch jobs where resources needed for tasks of different stages are significantly different

## 03. How it works



Coarse-Grained  
Resource management



Fine-Grained  
Resource management

## 03. SlotSharingGroup – Example

```
final StreamExecutionEnvironment env =  
    StreamExecutionEnvironment.getExecutionEnvironment();  
  
SlotSharingGroup ssgA = SlotSharingGroup.newBuilder("a")  
    .setCpuCores(1.0)  
    .setTaskHeapMemoryMB(100)  
    .build();  
  
SlotSharingGroup ssgB = SlotSharingGroup.newBuilder("b")  
    .setCpuCores(0.5)  
    .setTaskHeapMemoryMB(100)  
    .build();
```

## 03. SlotSharingGroup – Example

```
someStream.filter(...).slotSharingGroup("a")
    // Set the slot sharing group with name “a”
    .map(...).slotSharingGroup(ssgB);
    // Directly set the slot sharing group with name and resource.
```

```
env.registerSlotSharingGroup(ssgA);
    // Then register the resource of group “a”
```

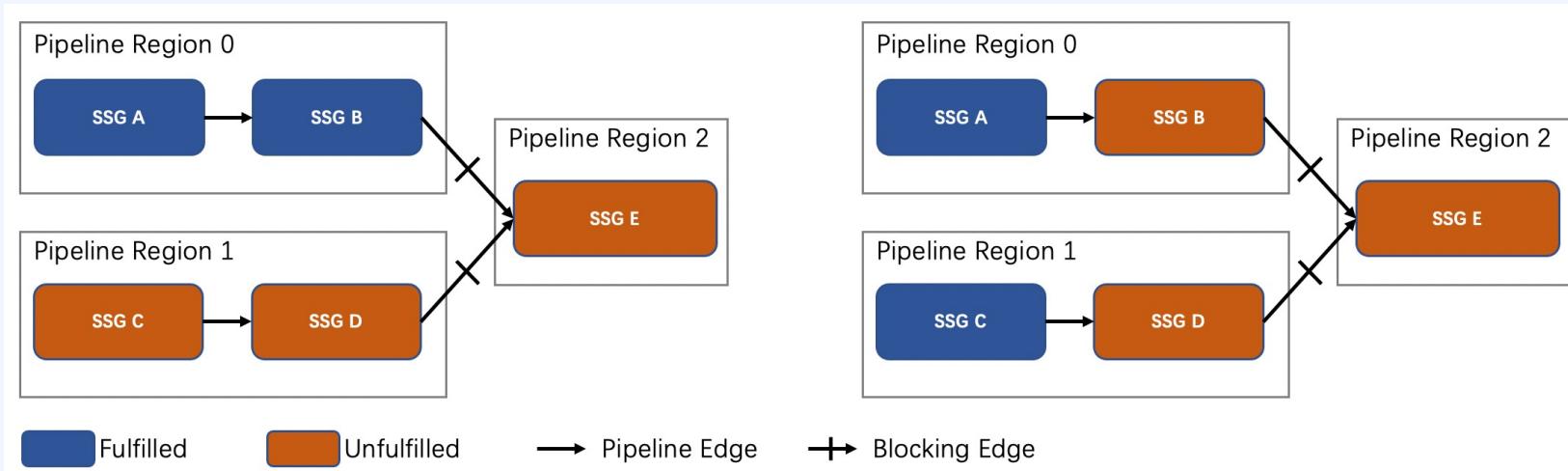
## 03. SlotSharingGroup build – Example

```
// Directly build a slot sharing group with specific resource
SlotSharingGroup ssgWithResource =
    SlotSharingGroup.newBuilder("ssg")
        .setCpuCores(1.0) // required
        .setTaskHeapMemoryMB(100) // required
        .setTaskOffHeapMemoryMB(50)
        .setManagedMemory(MemorySize.ofMebiBytes(200))
        .setExternalResource("gpu", 1.0)
        .build();
```

## 03. Limitations

- No support for the Elastic Scaling.
- Limited integration with Flink's Web UI.
- Limited integration with batch jobs.

## 03. FLINK-20865



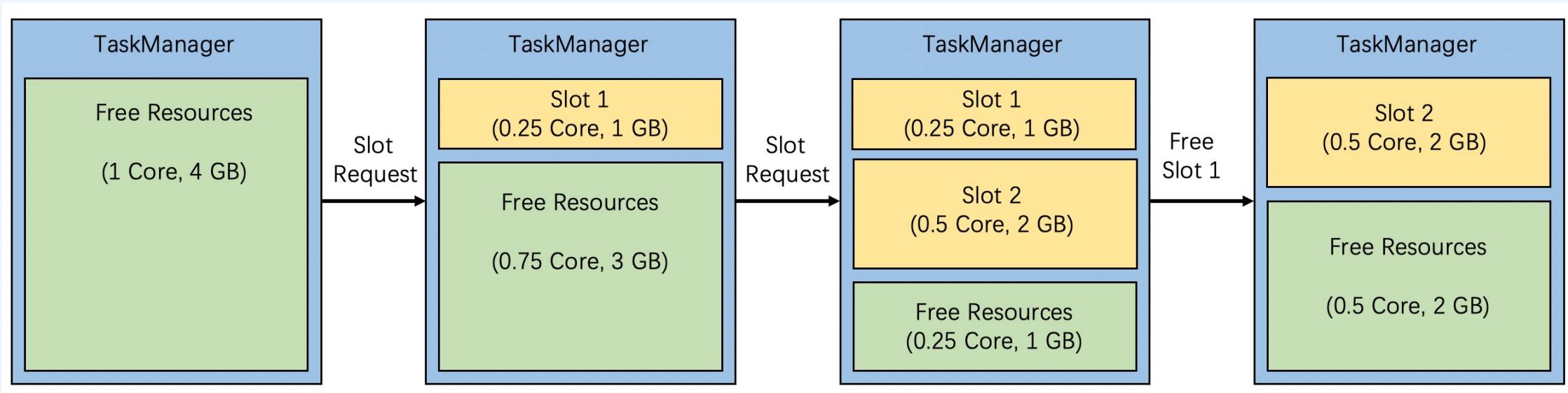
## 03. Limitations

- Hybrid resource requirements are not recommended.
- Slot allocation result might not be optimal.

## 03. Notice

- Setting the slot sharing group may change the performance.
- Slot sharing group will not restrict the scheduling of operators

## 03. Resource Allocation Strategy



# Chapter 7.

## 04. Metric Reporters

## 04. Metric Reporters

- conf/flink-conf.yaml
  - factory.class
  - interval
  - scope.delimiter
  - filter.includes
  - filter.excludes

## 04. Configuration- Example

```
metrics.reporters: my_jmx_reporter,my_other_reporter
```

```
metrics.reporter.my_jmx_reporter.factory.class:
```

```
    org.apache.flink.metrics.jmx.JMXReporterFactory
```

```
metrics.reporter.my_jmx_reporter.port: 9020-9040
```

```
metrics.reporter.my_jmx_reporter.scope.variables.excludes:
```

```
    job_id;task_attempt_num
```

```
metrics.reporter.my_jmx_reporter.scope.variables.additional:
```

```
    cluster_name:my_test_cluster,tag_name:tag_value
```

## 04. Configuration- Example

```
metrics.reporter.my_other_reporter.factory.class:  
    org.apache.flink.metrics.graphite.GraphiteReporterFactory  
metrics.reporter.my_other_reporter.host: 192.168.1.1  
metrics.reporter.my_other_reporter.port: 10000
```

## 04. Identifiers vs. tags

- Identifier-based reporters
  - job.MyJobName.numRestarts
- Tag-based reporter
  - jobName=MyJobName

## 04. Push vs. Pull

- Push-based reporters
  - Scheduled interface
- Pull-based reporters

## 04. Reporters - JMX

- Type: pull/tags
- Parameter
  - port
- Example
  - metrics.reporter.jmx.factory.class:  
org.apache.flink.metrics.jmx.JMXReporterFactory  
metrics.reporter.jmx.port: 8789
- Identified by
  - domain
  - list of key-properties

## 04. Reporters - InfluxDB

- Type: push/tags
- Parameter
  - connectTimeout
  - consistency
  - db, host, port
  - username, password
  - retentionPolicy

## 04. Reporters - InfluxDB

- Example
  - metrics.reporter.influxdb.factory.class:  
org.apache.flink.metrics.influxdb.InfluxdbReporterFactory
  - metrics.reporter.influxdb.scheme: http
  - metrics.reporter.influxdb.host: localhost
  - metrics.reporter.influxdb.port: 8086
  - metrics.reporter.influxdb.db: flink
  - metrics.reporter.influxdb.username: flink-metrics
  - metrics.reporter.influxdb.password: qwerty
  - metrics.reporter.influxdb.retentionPolicy: one\_hour

## 04. Reporters - InfluxDB

- Example
  - metrics.reporter.influxdb.consistency: ANY
  - metrics.reporter.influxdb.connectTimeout: 60000
  - metrics.reporter.influxdb.writeTimeout: 60000
  - metrics.reporter.influxdb.interval: 60 SECONDS
- Send metrics
  - using http protocol with specified retention policy

## 04. Reporters - InfluxDB

- Consistency with CAP
  - CA
    - ALL: Availability(X), Consistency(O)
    - ANY: Availability(O), Consistency(X)
    - ONE: Availability(O), Consistency(x)
    - QUORUM: Availability(-), Consistency(-)
  - P
    - ALL: Partition tolerance(X)

## 04. Reporters - Prometheus

- Type: pull/tags
- Parameter
  - port
  - filterLabelValueCharacters
- Example
  - metrics.reporter.prom.factory.class:  
org.apache.flink.metrics.prometheus.PrometheusReporterFactory

## 02. Reporters – Prometheus metric types

Flink	Prometheus
Counter	Gauge
Gauge	Gauge
Histogram	Summary
Meter	Gauge
Timer	Gauge/Histogram
-	Counter

## 04. Reporters - Prometheus

- All Flink metrics variables (see List of all Variables) are exported to Prometheus as labels.
  - job\_name=MyJob
  - host=localhost

## 04. Reporters - PrometheusPushGateway

- Type: push/tags
- Parameter
  - deleteOnShutdown
  - filterLabelValueCharacters
  - groupingKey
  - hostUrl
  - jobName
  - randomJobNameSuffix

## 04. Reporters - PrometheusPushGateway

- Example
  - metrics.reporter.promgateway.factory.class:  
org.apache.flink.metrics.prometheus.PrometheusPushGatewayReporte  
rFactory
  - metrics.reporter.promgateway.hostUrl: http://localhost:9091
  - metrics.reporter.promgateway.jobName: myJob
  - metrics.reporter.promgateway.randomJobNameSuffix: true
  - metrics.reporter.promgateway.deleteOnShutdown: false
  - metrics.reporter.promgateway.groupingKey: k1=v1;k2=v2
  - metrics.reporter.promgateway.interval: 60 SECONDS

## 04. PrometheusReporter vs. PrometheusPushGatewayReporter

- PrometheusReporter: pull
- PrometheusPushGatewayReporter: push

## 04. Metrics

- RichFunction
  - getRuntimeContext().getMetricGroup()  
-> MetricGroup object

## 04. Metric types

- Counter
- Gauge
- Histogram
- Meter

## 04. Metric types - Counter

```
class MyMapper extends RichMapFunction<String, String> {  
    private transient Counter counter;  
  
    @Override  
    public void open(Configuration config) {  
        this.counter = getRuntimeContext()  
            .getMetricGroup()  
            .counter("myCounter");  
    }  
    ...  
}
```

## 04. Metric types - Counter

```
class MyMapper extends RichMapFunction<String, String> {  
    private transient Counter counter;  
    ...  
    @Override  
    public String map(String value) throws Exception {  
        this.counter.inc();  
        return value;  
    }  
}
```

## 04. Metric types – Custom Counter

```
public class MyMapper extends RichMapFunction<String, String> {  
    private transient Counter counter;  
  
    @Override  
    public void open(Configuration config) {  
        this.counter = getRuntimeContext()  
            .getMetricGroup()  
            .counter("myCustomCounter", new CustomCounter());  
    }  
    ...  
}
```

## 04. Metric types – Custom Counter

```
public class CustomCounter implements Counter {  
    private final AtomicLong counter = new AtomicLong();  
    @Override  
    public void inc() {  
        counter.incrementAndGet();  
    }  
    @Override  
    public void inc(long n) {  
        counter.addAndGet(n);  
    }  
    ...
```

## 04. Metric types – Custom Counter

```
public class CustomCounter implements Counter {  
    ...  
    @Override  
    public void dec() {  
        counter.decrementAndGet();  
    }  
    @Override  
    public void dec(long n) {  
        counter.addAndGet(-n);  
    }  
    ...
```

## 04. Metric types – Custom Counter

```
public class CustomCounter implements Counter {  
    ...  
    @Override  
    public long getCount() {  
        return counter.get();  
    }  
}
```