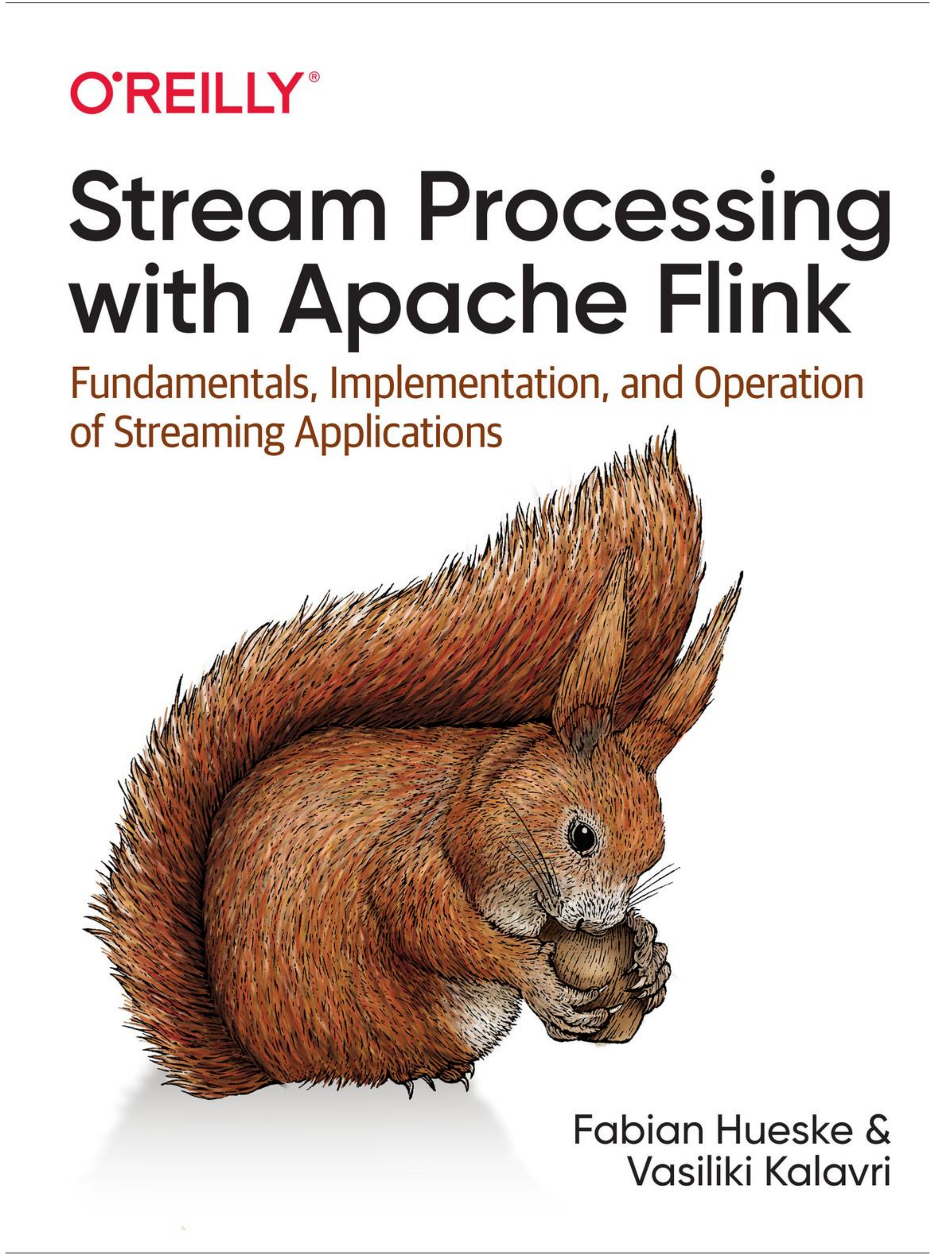
Stream Processing
Apache Flink

Amin FARVARDIN

Ph.D. Computer Science
University Paris Dauphine


O'REILLY®

Stream Processing with Apache Flink

Fundamentals, Implementation, and Operation of Streaming Applications

Fabian Hueske & Vasiliki Kalavri

Chapter 1

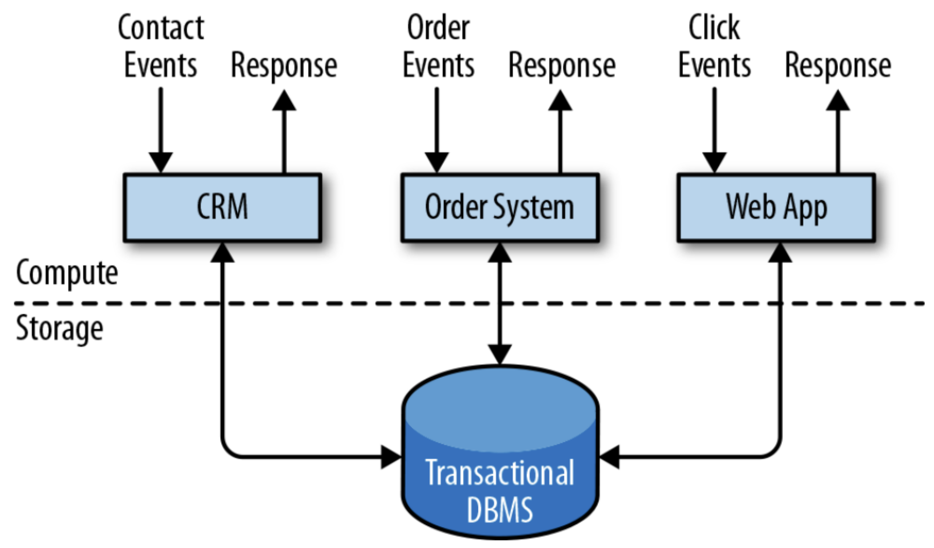
Introduction to Stateful Stream Processing

Traditional Data Infrastructures

- The traditional distinguishes two types of data processing:
    - Transactional processing
    - Analytical processing

Transactional Processing

- Use all kinds of applications for their business activities
- Separate tiers:
    - data processing
    - data storage
- Drawback:
    - When need to evolve or scale

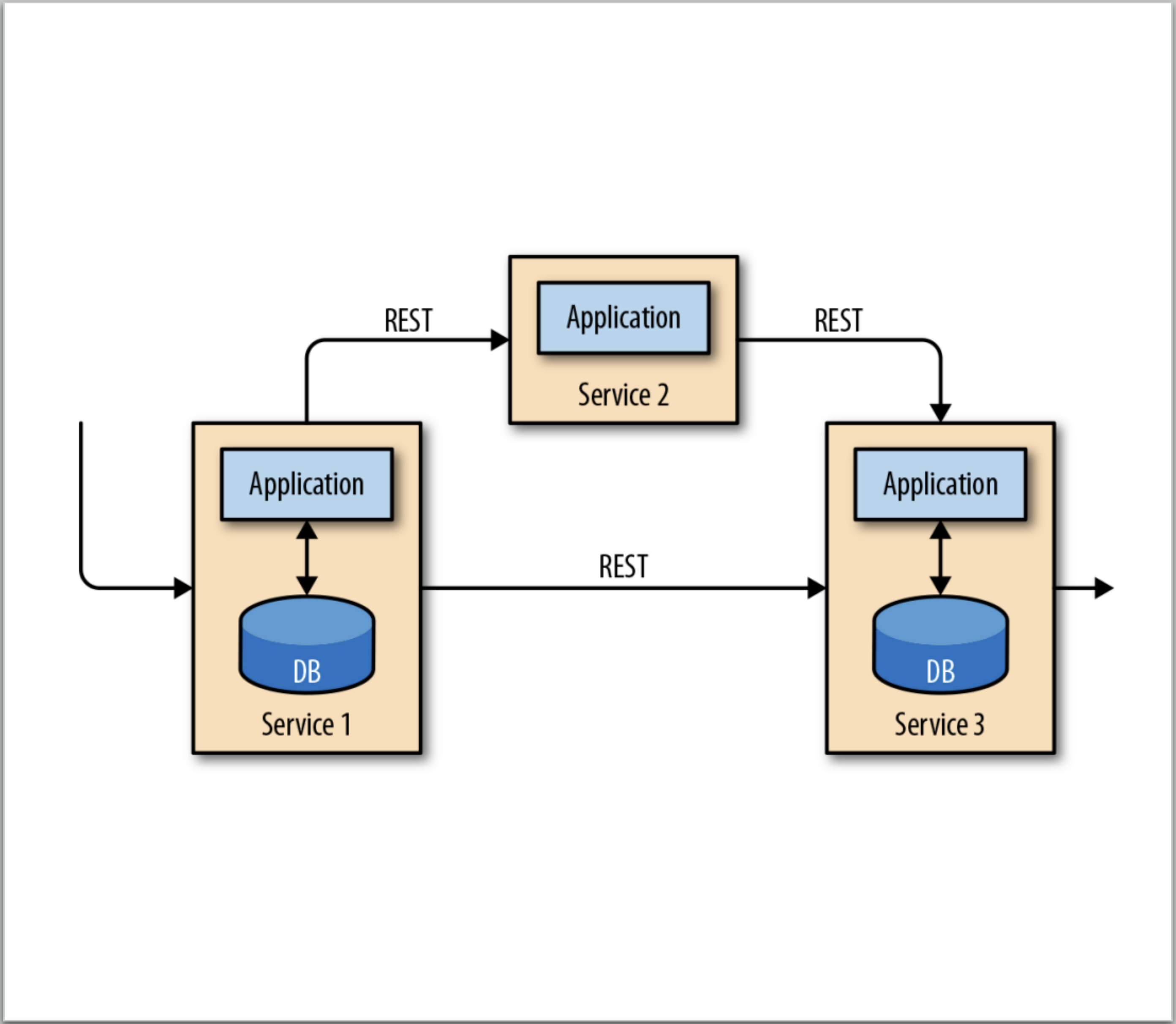


Traditional design of transactional applications that store data in a remote database system

Transactional Processing

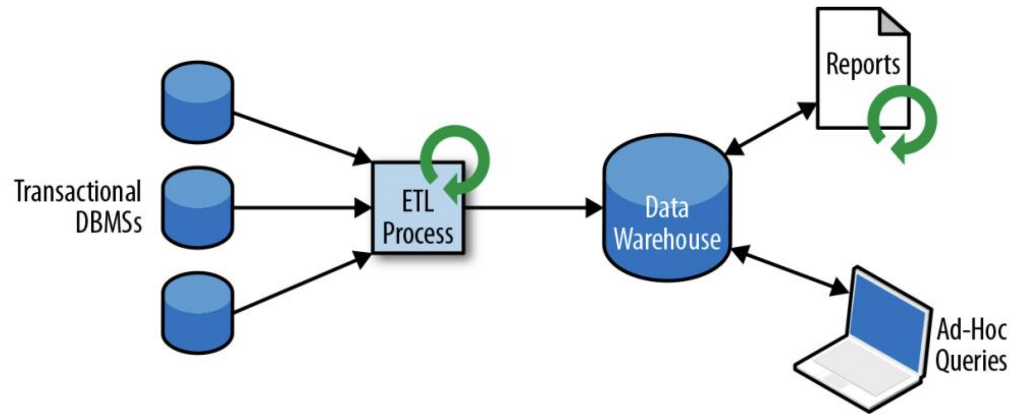
microservices design pattern

- Approach to overcoming the tight bundling of applications
- Designed as small, self-contained, and independent applications
- UNIX philosophy of doing a single thing and doing it well
- Communicate over standardized interfaces such as RESTful HTTP
- Can be implemented with a different technology stack
- Bundled and deployed in independent containers



Analytical Processing

- Generally, the data is stored in the various databases
- They distributed across several disconnected database systems
- Replicated to a data warehouse
- A dedicated datastore for analytical query workloads
- ETL
  - Can be quite complex
  - require sophisticated solutions to meet performance
  - Run periodically to keep data warehouse synchronized



Analytical Processing

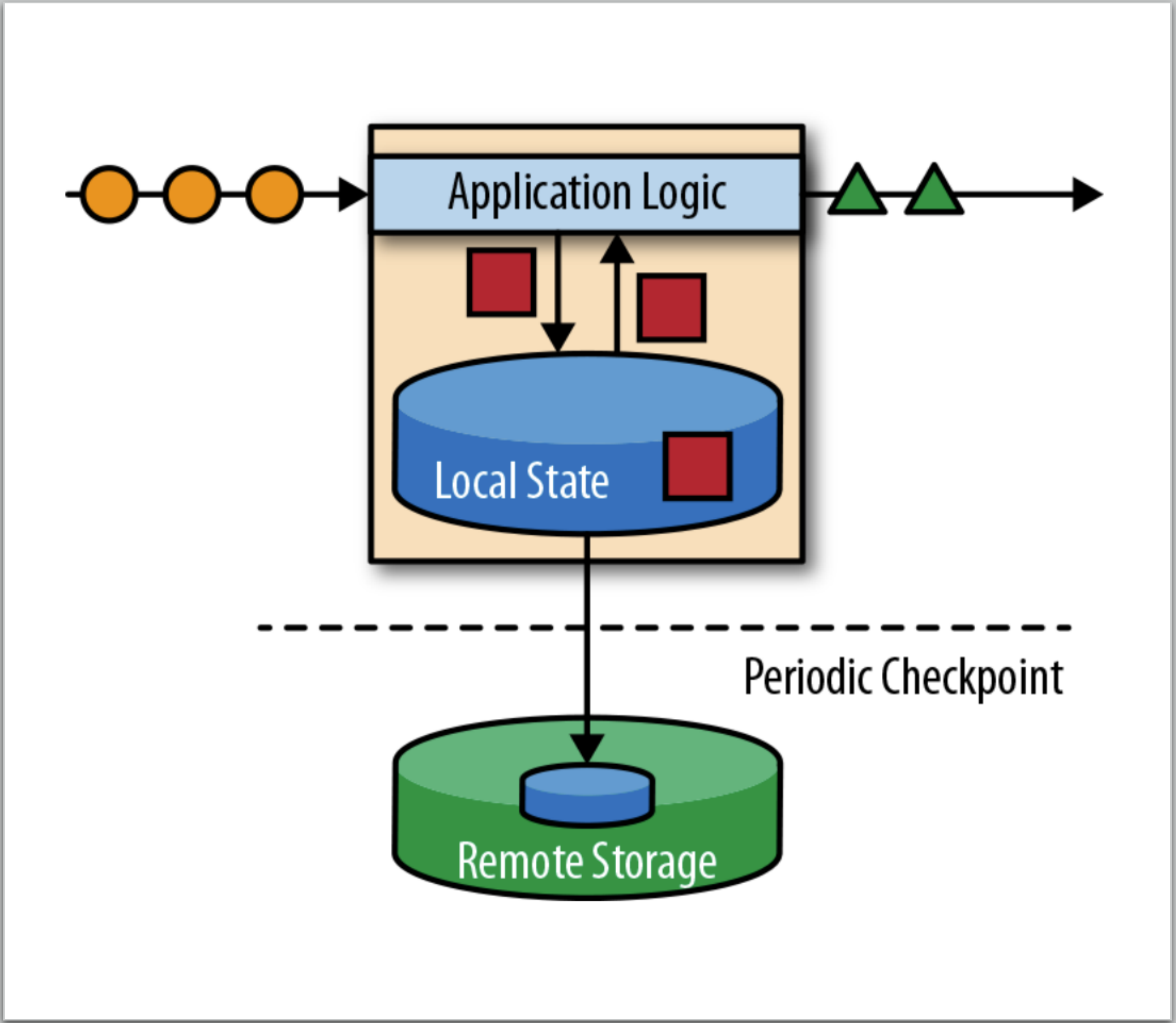
On Big Data – Hadoop ecosystem, etc.

- Today, many enterprises, store data in:
  - Hadoop's distributed filesystem (HDFS)
  - S3
  - other bulk datastores like Apache HBase
- Data queries with:
  - SQL-on-Hadoop engine (e.g., Apache Hive, Drill, Impala)

Note: The infrastructure remains basically the same as a traditional data warehouse architecture

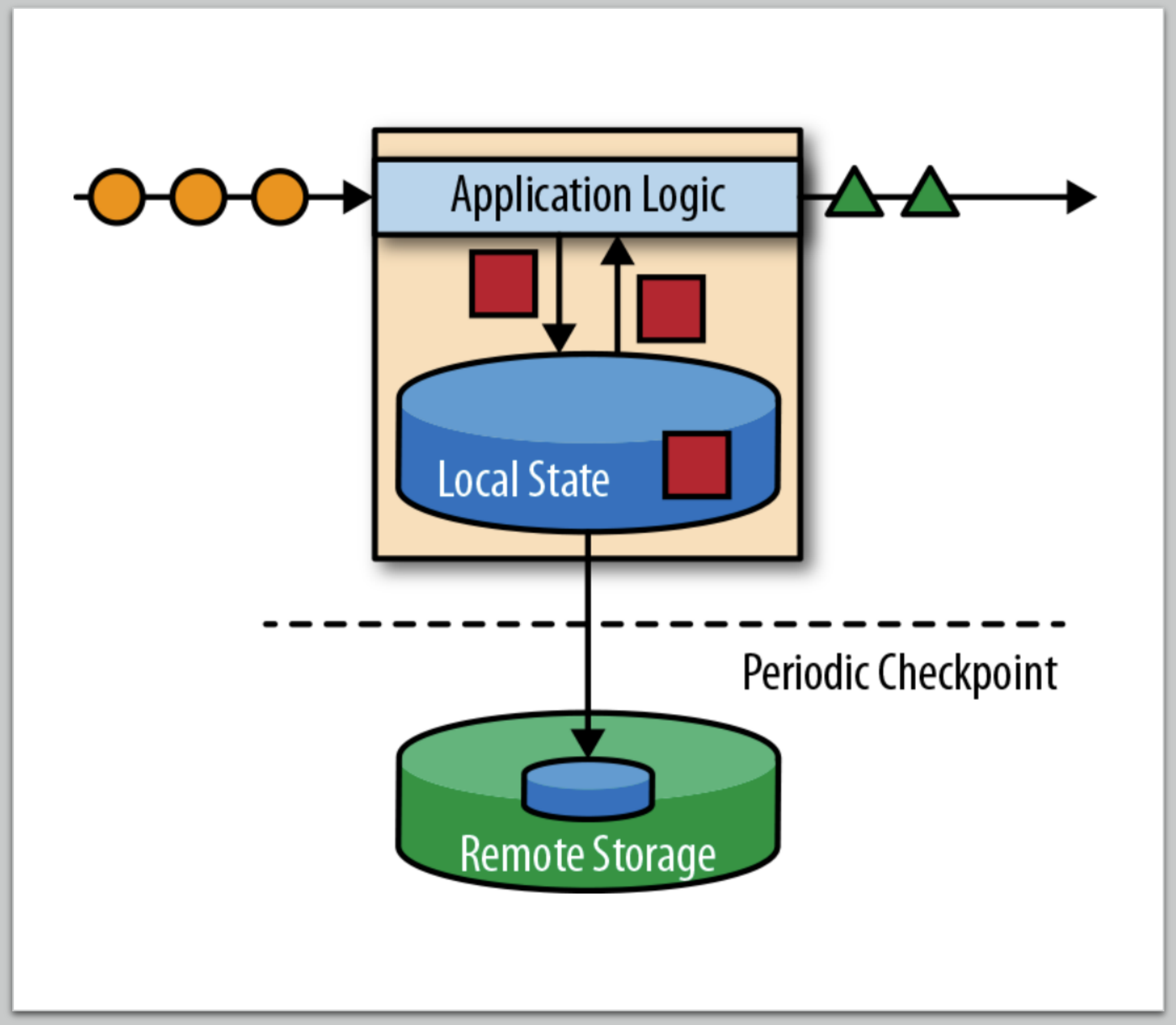
Stateful Stream Processing

- Virtually all data is created as continuous streams of events.
- All of user interactions are streams of events
- Stateful stream processing is
  - An application design pattern
  - For processing unbounded streams of events
  - It is applicable to many different use cases



Stateful Stream Processing

- Stateful Stream Processing characteristics:
    - processes a stream of events
    - ability to store and access intermediate data
- Apache Flink is a stateful streaming application
    - stores the application state locally in memory or in an embedded database
    - Since it is a distributed system
    - a consistent checkpoint of the application state to a remote and durable storage



Stateful Stream Processing

- Often ingest their incoming events from an event log
- An event log stores and distributes event streams
- Events are written to a durable, append-only log
- the order of written events cannot be changed
- A stream that is written to an event log can be read many times by the same or different consumers
- events are always published to all consumers in exactly the same order
- Apache Kafka the most popular event log systems

Stateful Stream Processing

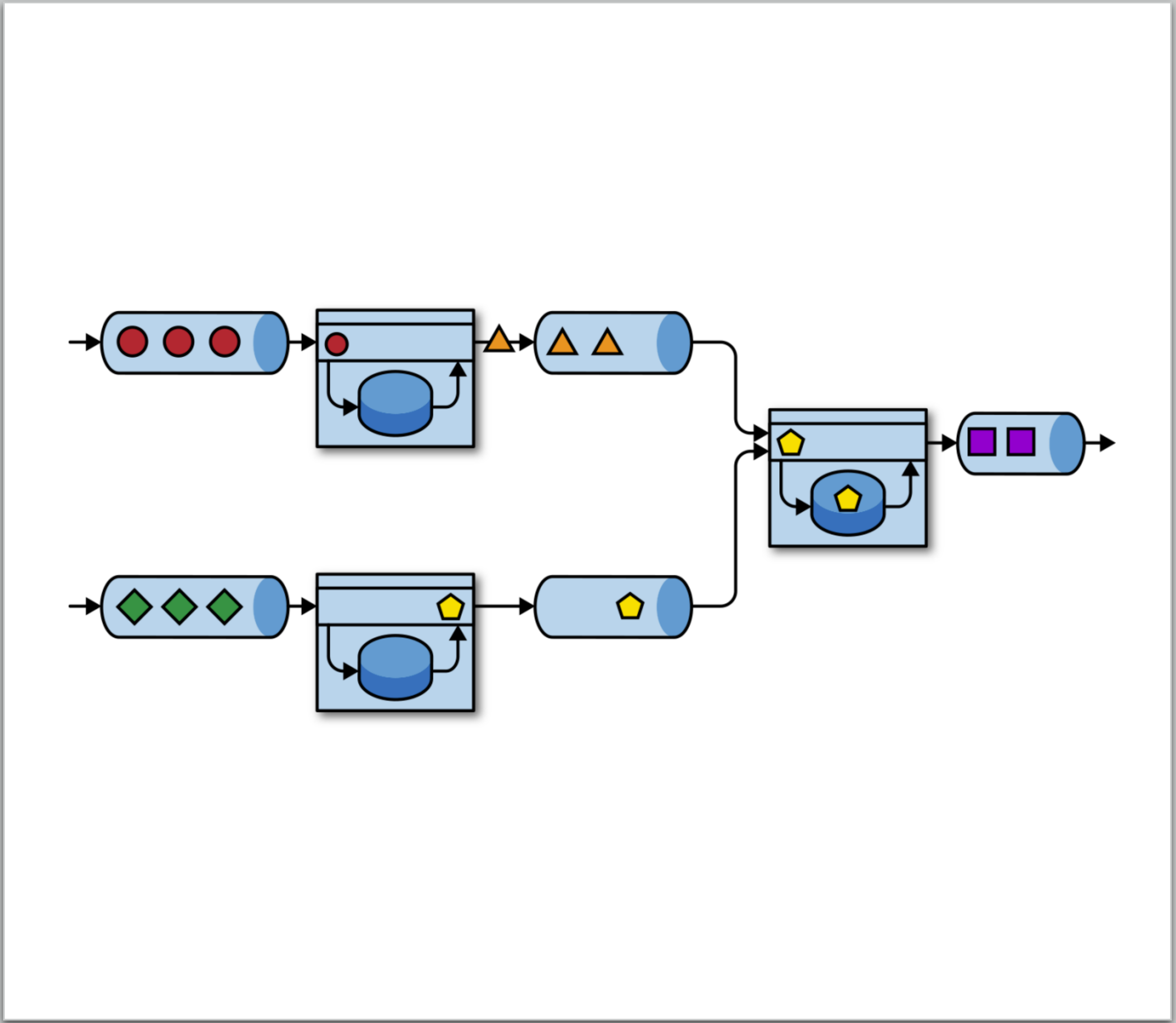
- Three classes of applications that are commonly implemented using stateful stream processing:
    1. Event-Driven applications
    2. Data Pipeline applications
    3. Data Analytics applications

Event-Driven Applications

- Ingest event streams and process the events with application-specific business logic
- An event-driven application can trigger actions such as sending an alert or an email or write events to an outgoing event stream to be consumed by another event-driven application.
- Typical use cases for event-driven applications include:
  - Real-time recommendations (e.g., for recommending products while customers browse a retailer’s website)
  - Pattern detection or complex event processing (e.g., for fraud detection in credit card transactions)
  - Anomaly detection (e.g., to detect attempts to intrude a computer network)

Event-Driven Applications

- It’s an evolution of microservices
- They communicate via event logs instead of REST calls
- hold application data as local state instead of writing it to and reading it from an external datastore (i.e., a relational database or key-value store)
- One application emits its output to an event log and another application consumes the events
- Decouples senders and receivers and provides asynchronous, nonblocking event transfer
- Each application can be stateful and can locally manage its own state without accessing external datastores
- Applications can also be individually operated and scaled.



Event-Driven Applications

Several benefits compared to transactional applications or microservices

- Local state access provides very good performance compared to reading and writing queries against remote datastores
- Scaling and fault tolerance are handled by the stream processor
- by leveraging an event log (as input source) the complete input of an application is reliably stored and can be certainly replayed
- Flink can reset the state of an application to a previous savepoint,
- **Exactly-once state** consistency and the ability to scale an application,

Data Pipelines

- Many different datastores; that store data in different formats and data structures,
- Due to this replication of data, the data stores must be kept in sync
- A traditional approach to synchronize data in different storage systems is periodic ETL jobs.
  - But they do not meet the latency requirements for many use cases
- An alternative is to use an event log to distribute updates,
- Ingesting, transforming, and inserting data with low latency
- Data pipelines must be able to process large amounts of data in a short time

Streaming Analytics

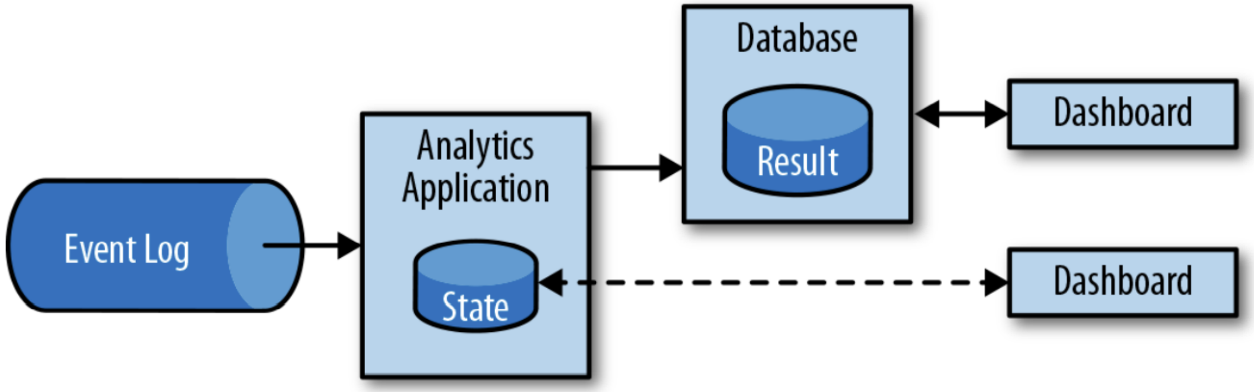
- Instead of waiting to be periodically an ETL triggered
  - a streaming analytics application continuously ingests streams of events and updates its result by incorporating the latest events with low latency.
  - This is similar to the maintenance techniques database systems use to update materialized views.
  - Typically, streaming applications store their result in an external data store that supports efficient updates
  - The live updated results of a streaming analytics application can be used to power dashboard applications

Streaming Analytics

- So, much shorter time needed for an event to be incorporated into an analytics result,

Advantage of streaming analytics applications:

- Traditional analytics pipelines consist of several individual components such as
    - an ETL process,
    - a storage system,
    - a data processor and scheduler to trigger jobs or queries.
- A stream processor that runs a stateful streaming application takes care of all these processing steps, including
    - event ingestion,
    - continuous computation including state maintenance, and updating the results.
    - It can recover from failures with exactly-once state consistency guarantees
    - It can adjust the compute resources of an application.



Streaming Analytics

Use cases

Streaming analytics applications are commonly used for:

- Monitoring the quality of cellphone networks
- Analyzing user behavior in mobile applications
- Ad-hoc analysis of live data in consumer technology

Apache Flink

- It is a third-generation distributed stream processor with a competitive feature set.
- It provides accurate stream processing with high throughput and low latency at scale.
- In particular, the following features make Flink stand out:
  - Event-time and processing-time semantics.
    - Event-time semantics provide consistent and accurate results despite out-of-order events.
    - Processing-time semantics can be used for applications with very low latency requirements.
  - **Exactly-once state consistency guarantees**.
  - **Millisecond latencies** while processing millions of events per second. Flink applications can be scaled to run on thousands of cores.

Apache Flink

- Layered APIs with varying tradeoffs for expressiveness and ease of use.
- Connectors to the most commonly used storage systems such as
  - Apache Kafka,
  - Apache Cassandra,
  - Elasticsearch,
  - JDBC,
  - Kinesis, and
  - (distributed) filesystems such as HDFS and S3.
- Ability to run streaming applications 24/7 with very little downtime due to its highly available setup (no single point of failure),
- tight integration with Kubernetes, YARN, and Apache Mesos,

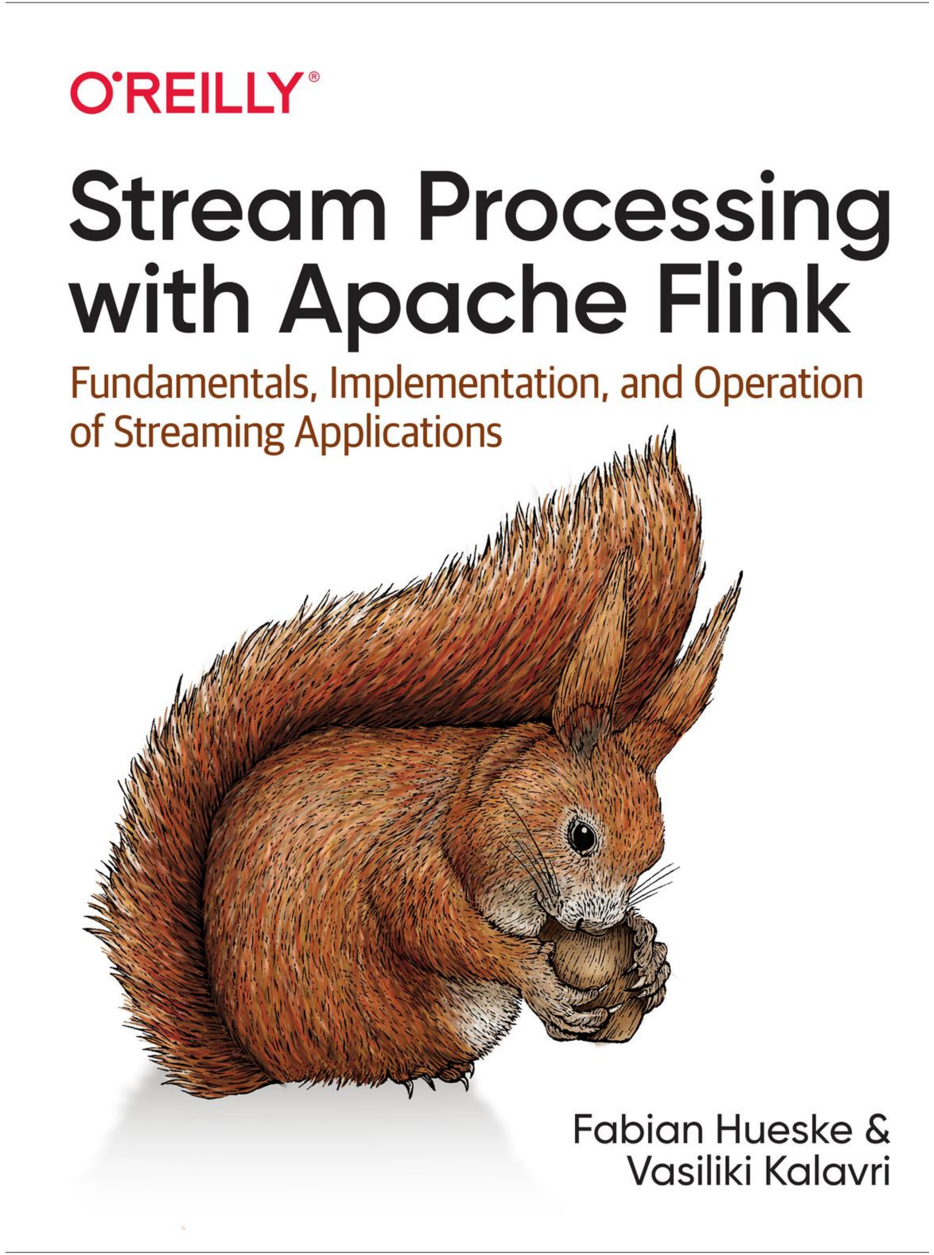
Apache Flink

- fast recovery from failures,
- the ability to dynamically scale jobs.
- Ability to update the application code of jobs and migrate jobs to different Flink clusters without losing the state of the application.
- Detailed and customizable collection of system and application metrics to identify and react to problems ahead of time.
- Flink is also a full-fledged batch processor
- Flink is a very developer-friendly framework due to its easy-to-use APIs
- whole Flink system in a single JVM process
- can be used to run and debug Flink jobs within an IDE.

Hands-on Practice

Running Your First Flink Application

http://bit.ly/3iPLygG



O'REILLY

Stream Processing with Apache Flink

Fundamentals, Implementation, and Operation of Streaming Applications

Fabian Hueske & Vasiliki Kalavri

Chapter 2

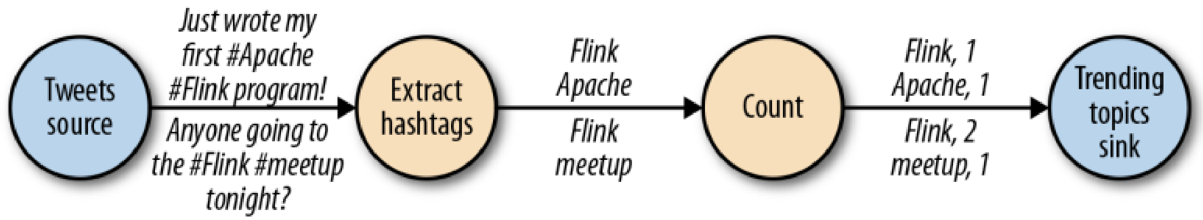
Stream Processing Fundamentals

Introduction to Dataflow Programming

Dataflow Graphs

Logical DG

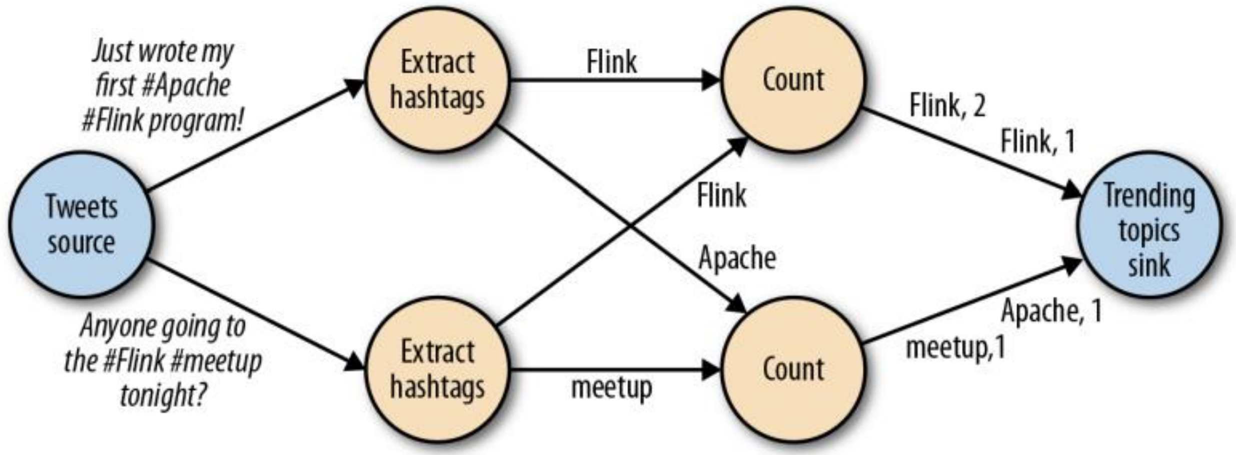
- Represented as directed graphs:
  - nodes are called operators and represent computations
  - edges represent data dependencies
- Operators are the basic functional units of a dataflow application.
- They consume data from inputs, perform a computation on them, and produce data to outputs for further processing.
- Operators without input ports are called data sources
- Operators without output ports are called data sinks.
- A dataflow graph must have at least one data source and one data sink.
- Logical DG, convey a high-level view of the computation logic



A logical dataflow graph to continuously count hashtags (nodes represent operators and edges denote data dependencies)

Dataflow Graphs
Physical DG

- specifies in detail how the program is executed
- in the physical dataflow, the nodes are tasks
- The “Extract hashtags" and “Count" operators have two parallel operator tasks, each performing a computation on a subset of the input data.



A physical dataflow plan for counting hashtags (nodes represent tasks)

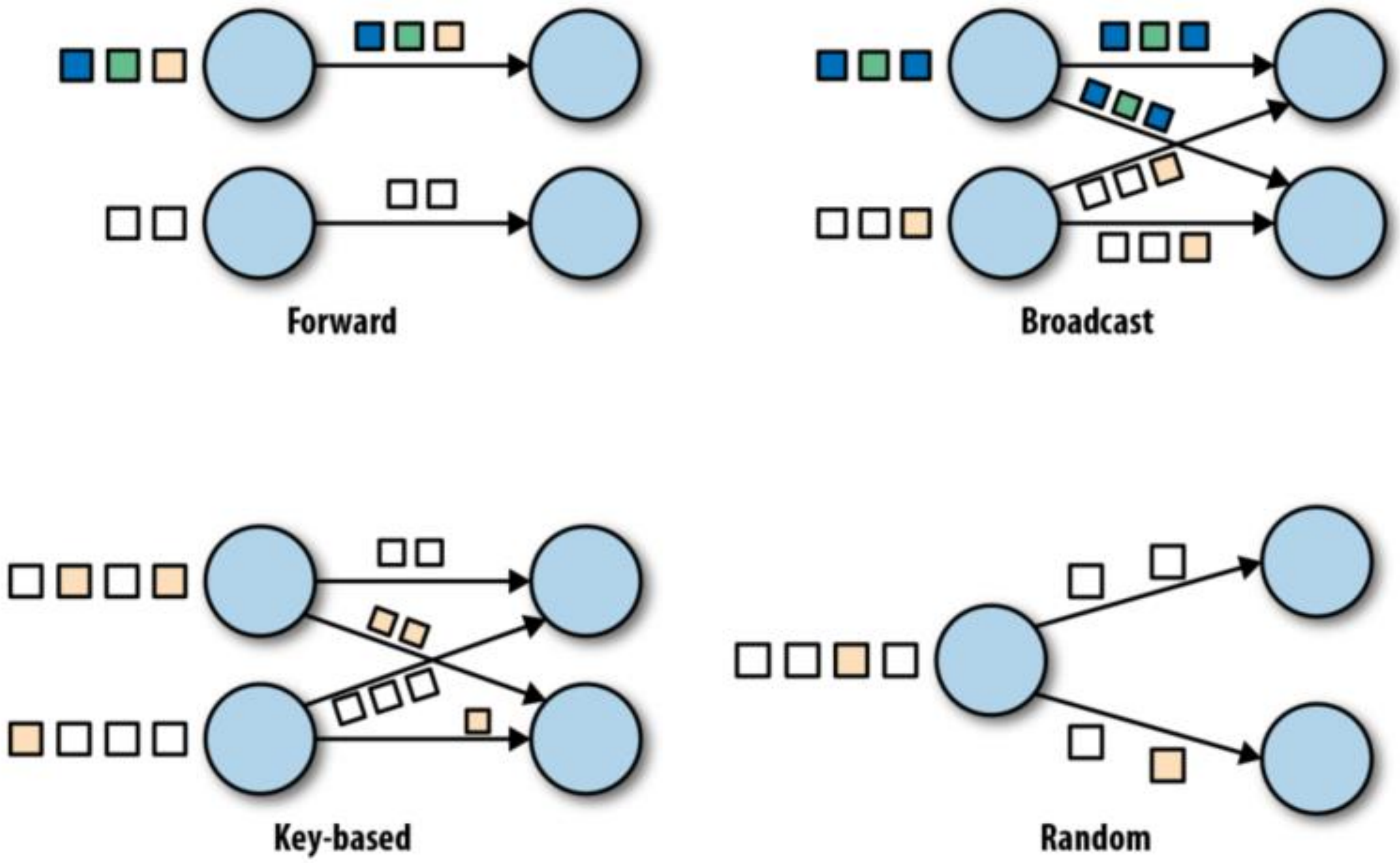
Parallelism

1. Data parallelism:
   - Definition: Partition input data and have tasks of the same operation execute on the data subsets in parallel
   - it allows for processing large volumes of data and spreading the computation load across several computing nodes

2. Task Parallelism
   - Definition: Having tasks from different operators performing computations on the same or different data in parallel.
   - Using task parallelism, you can better utilize the computing resources of a cluster

Data Exchange Strategies

How data items are assigned to tasks in a physical dataflow graph



Processing Streams in Parallel

Let’s define the term data stream:

- A data stream is a potentially unbounded sequence of events
- Events in a data stream can represent:
    - monitoring data,
    - sensor measurements,
    - credit card transactions,
    - weather station observations,
    - online user interactions,
    - web searches, etc.

Processing Streams in Parallel

Latency and Throughput

- Latency:
  - Batch applications:. Total execution time
  - Stream applications:.
    - Run continuously
    - Input potentially unbounded
    - **There is no notion of Total execution time**
    - Must provide results for incoming data as fast as possible

Stream application performance depends on terms of latency and throughput

Processing Streams in Parallel

Latency

- Ex: The service latency in a coffee shop:
  - This is the time you spend in the coffee shop; from the moment you enter until you have your first sip of coffee.

- Low latency is a key characteristic of stream processing
- It enables what we call real-time applications
- Latency is measured in units of time, such as milliseconds
- Average latency, maximum latency, or percentile latency
- Apache Flink, can offer latencies as low as a few milliseconds

Processing Streams in Parallel

Throughput

- Ex: Coffee shop:
  - If the shop is open from 7 a.m. to 7 p.m. and it serves 600 customers in one day, then its average throughput would be 50 customers per hour.
  - Then, While you want **latency to be as low as possible**, you generally want **throughput to be as high as possible**.

- A measure of the system's processing capacity, its **rate of processing**

Latency and throughput are not independent metrics

Operations on Data Streams

- Stream processing engines usually provide a set of built-in operations to ingest, transform, and output streams
- Operations can be either **stateless** or **stateful**

Operations on Data Streams
Stateless operations

- Do not maintain any internal state
- That is, the processing of an event does not depend on any events seen in the past and no history is kept
- It’s easy to parallelize
- Moreover, in the case of a failure, a stateless operator can be simply restarted and continue processing from where it left off

Operations on Data Streams
Stateful operations

- may maintain information about the events they have received before
- State can be updated by incoming events
- They may used in the processing logic of future events

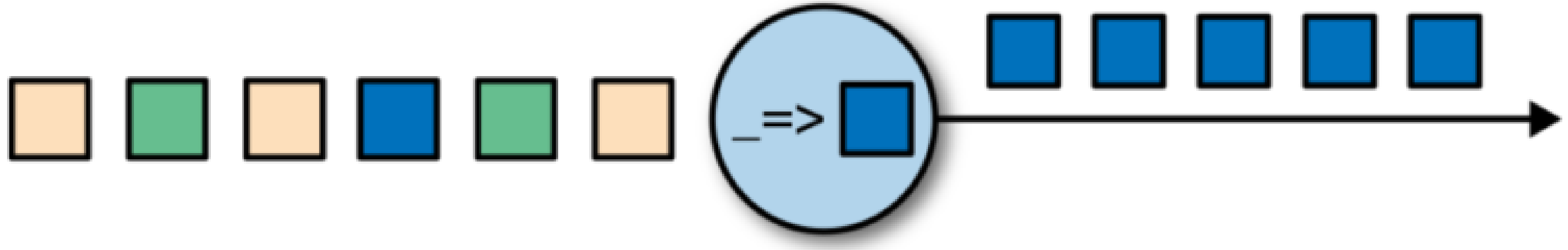
- Stateful stream processing **applications** are:
  - more challenging to parallelize and operate in a fault-tolerant manner
  - State needs to be efficiently partitioned and reliably recovered in the case of failures

DATA INGESTION AND DATA EGRESS

- Data ingestion and data egress operations allow the stream processor to communicate with external systems
- Data ingestion is the operation of fetching raw data from external sources and converting it into a format suitable for processing.
- Operators that implement data ingestion logic are called data sources.
- A data source can ingest data from:
    - a TCP socket, a file, a Kafka topic, or a sensor data interface.
- Data egress is the operation of producing output in a form suitable for consumption by external systems.
- Operators that perform data egress are called **data sinks** and examples include files, databases, message queues, and monitoring interfaces.

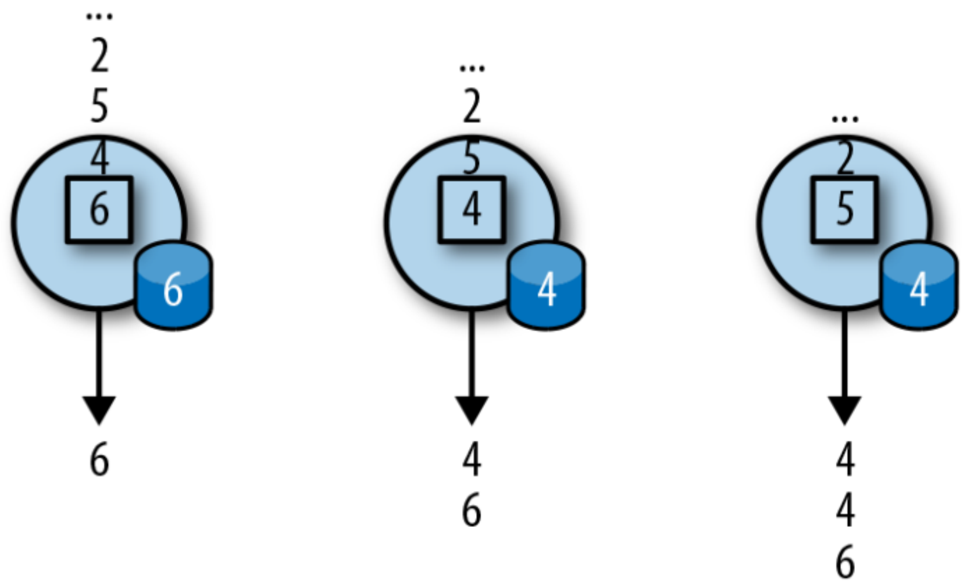
TRANSFORMATION OPERATIONS

- Single-pass operations that process each event independently
- Consume one event after the other and apply some transformation to the event data, producing a new output stream
- Operators can accept multiple inputs and produce multiple output streams
- They can also modify the structure of the dataflow graph by either:
    - splitting a stream into multiple streams or
    - merging streams into a single flow



ROLLING AGGREGATIONS

- Continuously updated for each input event:
  - Sum, minimum, and maximum
- Stateful and combine the current state with the incoming event
- **Note:** So, the aggregation function must be associative and commutative
- Otherwise, the operator would have to store the complete stream history.



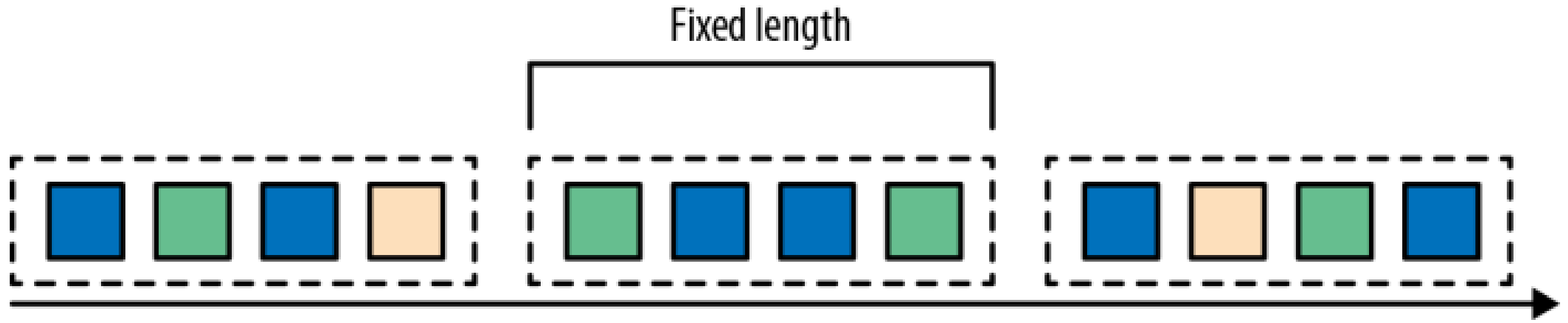
WINDOW OPERATIONS

- Continuously create finite sets of events called buckets from an unbounded event stream and let us perform computations on these finite sets.
- Events are usually assigned to buckets based on data properties or based on time.
- Window operator semantics need to determine both
  - how events are assigned to buckets and
  - how often the window produces a result.
- The behavior of windows is defined by a set of policies
  - when new buckets are created,
  - which events are assigned to which buckets, and
  - when the contents of a bucket get evaluated.
- Policies can be based on
  - time (e.g., events received in the last five seconds)
  - count (e.g., the last one hundred events)
  - a data property

WINDOW OPERATIONS

Tumbling windows

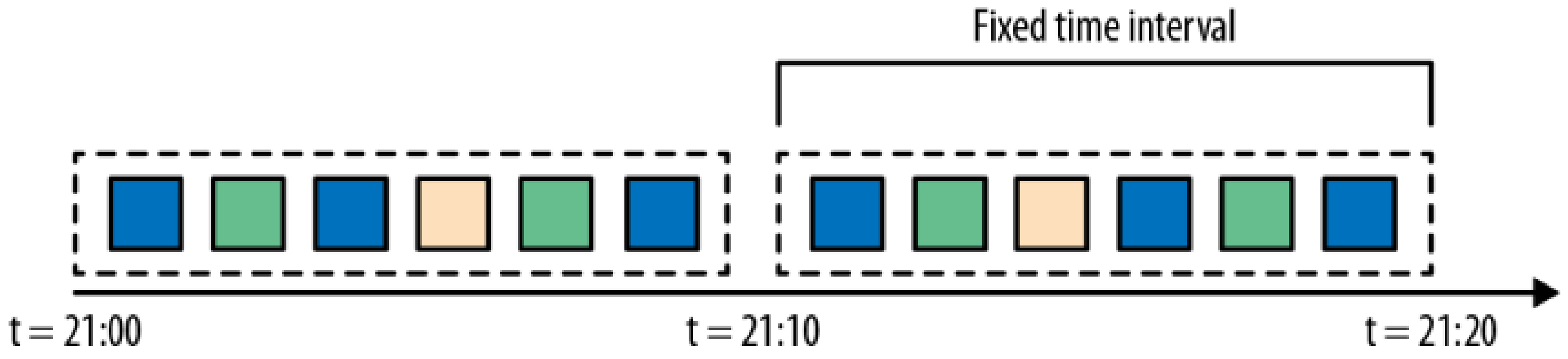
- Assign events into nonoverlapping buckets of fixed size
- **Count-based tumbling windows**
  - define how many events are collected before triggering evaluation



WINDOW OPERATIONS

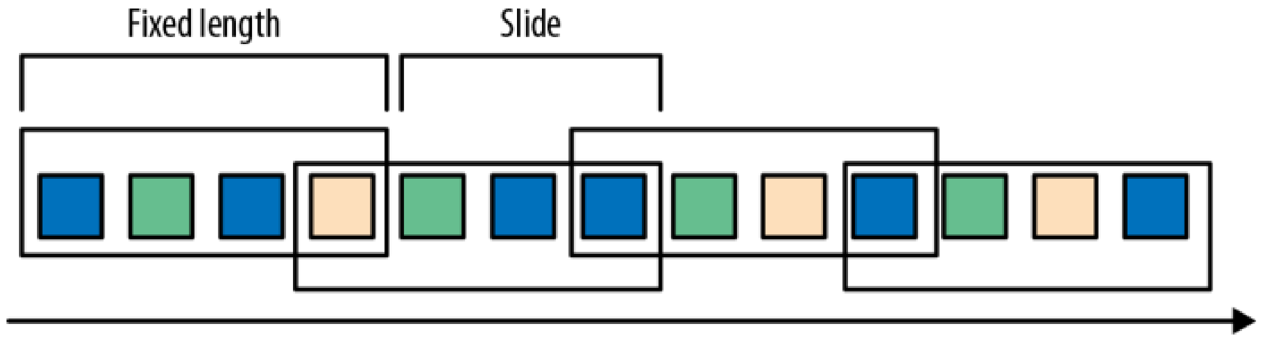
Tumbling windows

- **Time-based tumbling windows**
    - Define a time interval during which events are buffered in the bucket
- Ex: gathers events into buckets and triggers computation every 10 minutes



WINDOW OPERATIONS
Sliding windows

- **Sliding windows** assign events into overlapping buckets of fixed size.
    - An event might belong to multiple buckets
- Sliding windows by providing
    - length size
    - Slide size.
- The slide value defines the interval at which a new bucket is created

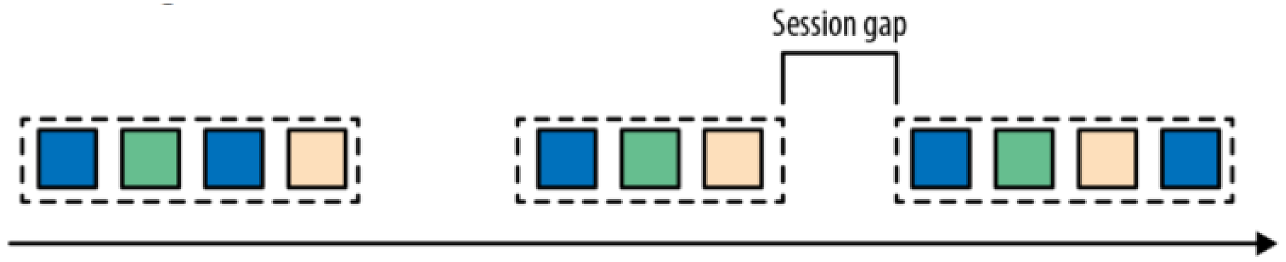


Sliding count-based window with a length of four events and a slide of three events

WINDOW OPERATIONS

Session windows

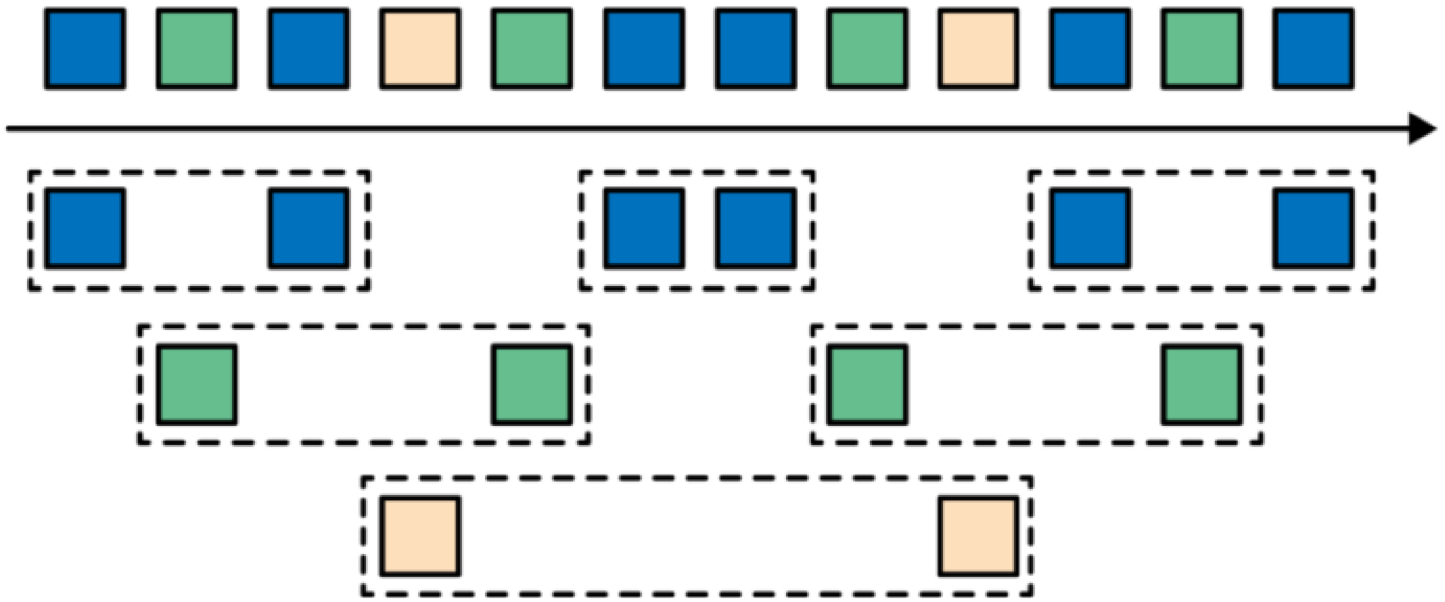
- **Session windows**: Useful in common real-world scenarios
  - A series of events happening in adjacent times followed by a period of inactivity
  - To group together events that originate from the same period of user activity or session
  - Ex: analyzes online user behavior
- Sessions are comprised of a series of events happening in adjacent times followed by a period of inactivity
- The length of a session is not defined beforehand so,
  - Tumbling and Sliding windows cannot be applied
- Ex: user interactions with a series of news articles one after the other
- Session windows group events in sessions based on a session gap value
- the time of inactivity is considered as a session closed



Session window

WINDOW OPERATIONS
Parallel windows

- To partition a stream into multiple logical streams
  - Ex: Receiving measurements from different sensors need to group the stream by sensor ID before applying a window computation



A parallel count-based tumbling window of length 2

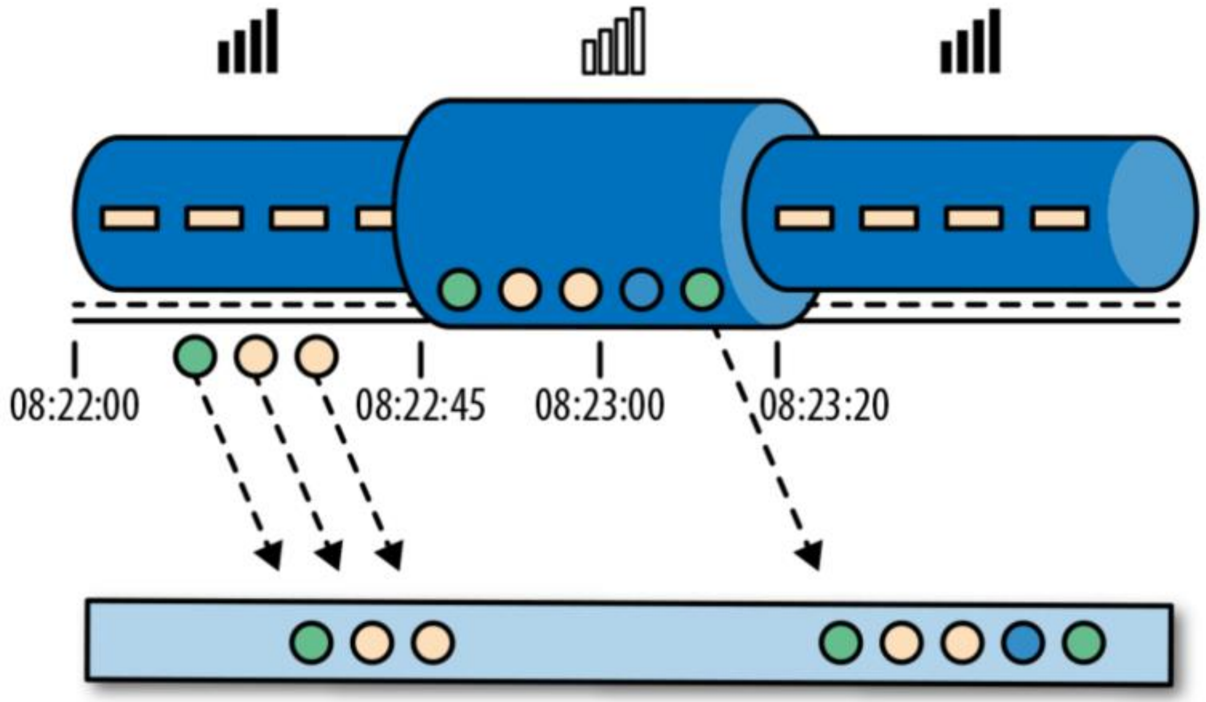
WINDOW OPERATIONS
General concepts

- Window operations are closely related to two dominant concepts in stream processing:
  - Time semantics (is perhaps the most important aspect of stream processing)
  - State management
- Even though low latency is an attractive feature of stream processing, its true value is way beyond just fast analytics.
- Real-world systems, networks, and communication channels are far from perfect,
  - streaming data can **often be delayed** or **arrive out of order**.
- Streaming applications that **process events as they are produced** should also be able to
  - process historical events in the same way,
  - enabling offline analytics or even time travel analyses
- Guard state against failures is crucial
- All the window types so far, do buffer data before producing a result to maintain state
- Reliably recovered under failures and that your system can guarantee accurate results even if things break
  - Considering that they might run for several days, months, or even years

Time Semantics

What Does One Minute Mean in Stream Processing?

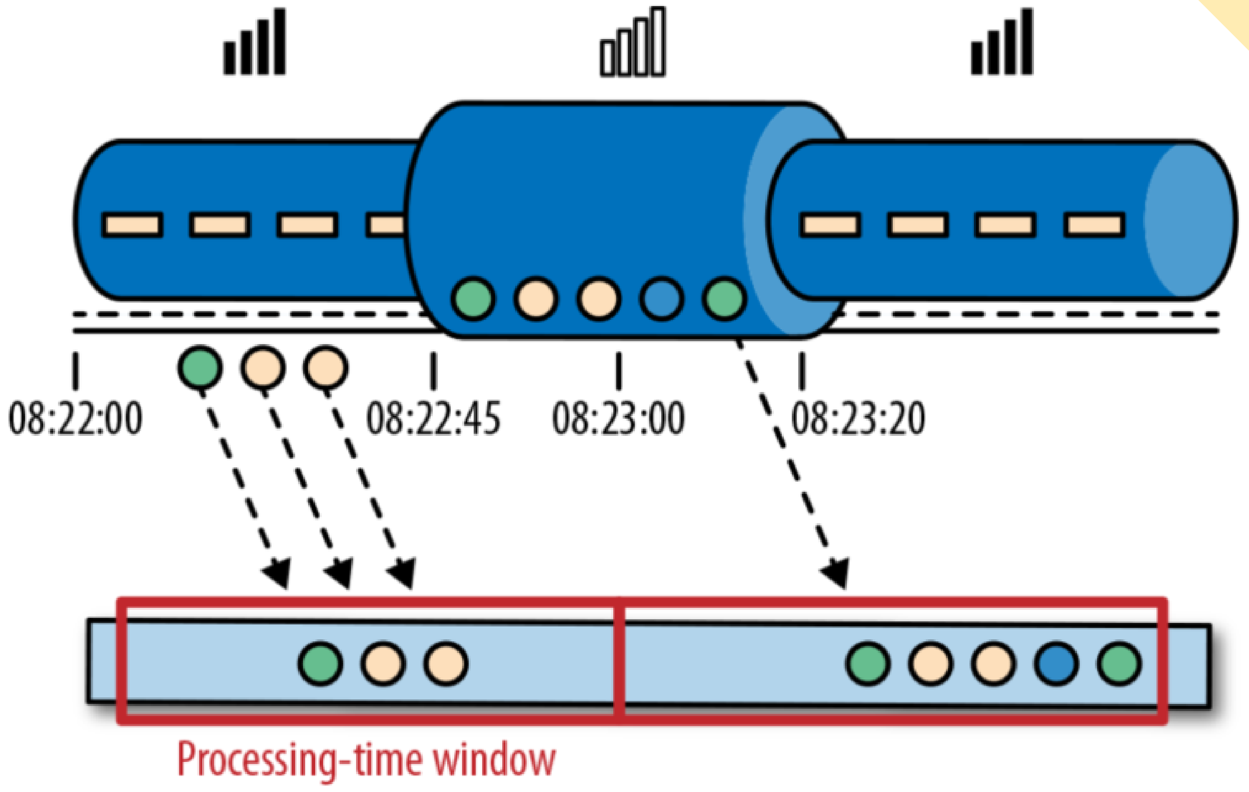
- Processing time
- Event Time



An application receiving online mobile game events played on the subway would experience a gap when the network connection is lost, but events are buffered in the player's phone and delivered when the connection is restored

Time Semantics
Processing Time

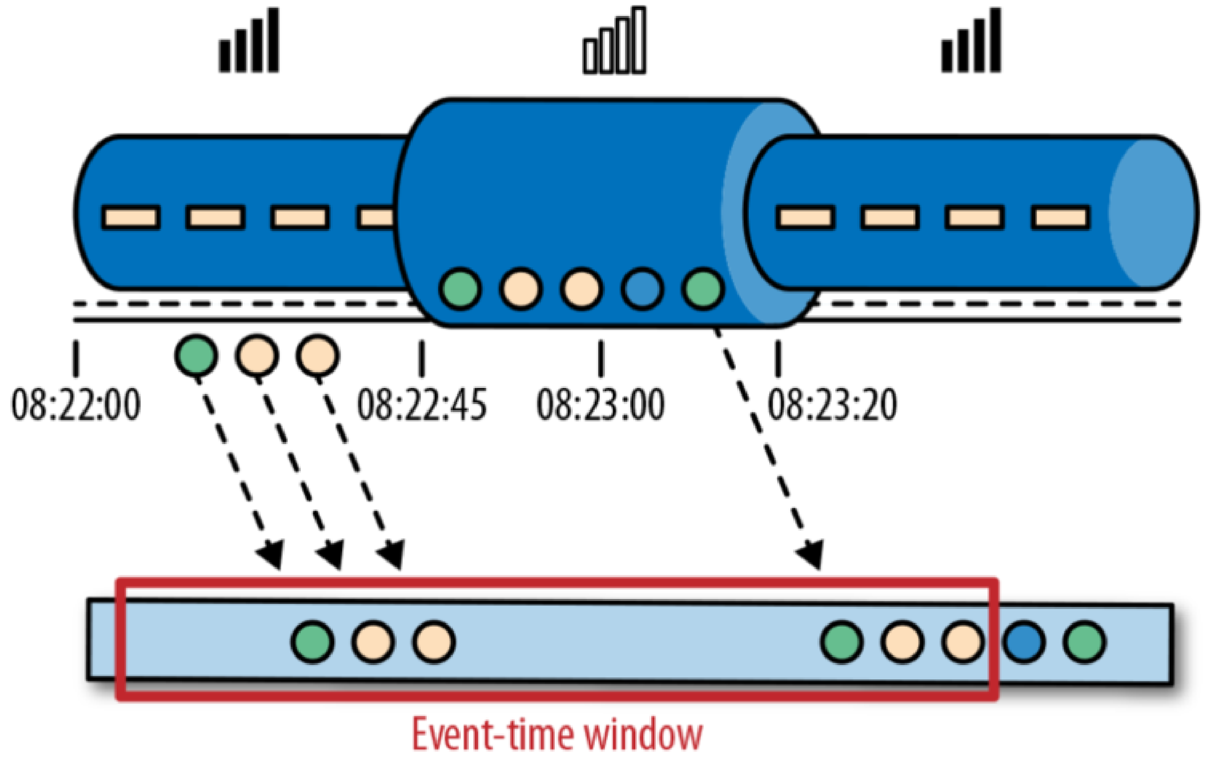
- The time of the local clock on the machine where the operator processing the stream is being executed.
- A processing-time window includes all events that happen to have arrived at the window operator within a time period, as measured by **the wall clock** of its machine.
- In Alice's case, a processing-time window would continue counting time when her phone gets disconnected, thus not accounting for her game activity during that time



A processing-time window continues counting time even after Alice's phone gets disconnected

Time Semantics
Event Time

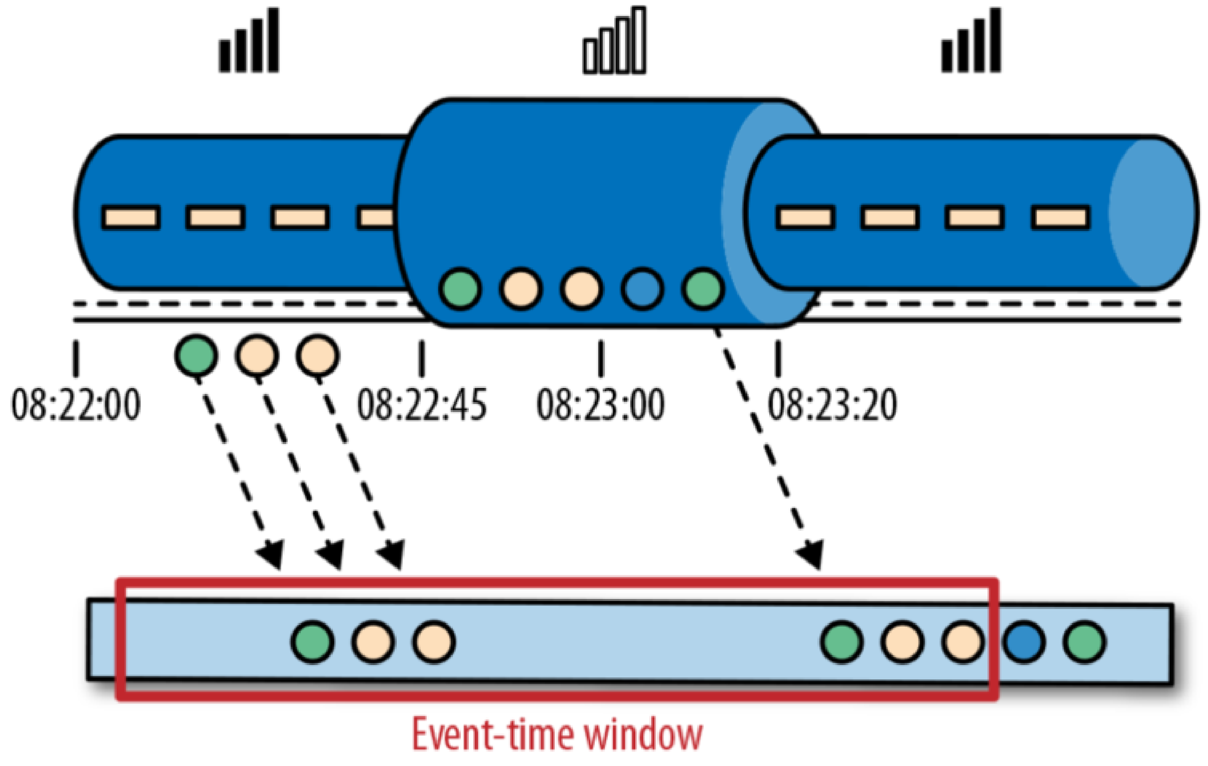
- Event time is the time when an event in the stream actually happened.
- Event time is based on a ***timestamp*** that is attached to the events of the stream
- Timestamps usually exist inside the event data before they enter the processing pipeline (e.g., the event creation time).
- An event-time window would correctly place events in a window, reflecting the reality of how things happened, even though some events were *delayed.*
- Event time completely decouples the processing speed from the results



Event time correctly places events in a window, reflecting the reality of how things happened

Time Semantics
Event Time

- Operations based on event time are predictable and their results are deterministic
- An event time window computation will yield the same result no matter how fast the stream is processed or when the events arrive at the operator
- The ubiquitous problem of out-of-order data can also be solved with event time
- The determinism of timestamps in gives you the ability to *fast forward* the past (Bob)
- So, once the program catches up with the events happening now, it can continue as a real-time application using exactly the same program logic



Event time correctly places events in a window, reflecting the reality of how things happened

Watermarks

- So far, we have overlooked one very important aspect:
    - how do we decide when to trigger an event-time window?
    - how long do we have to wait before we can be certain that we have received all events that happened before a certain point of time?
    - how do we even know that data will be delayed?
- We learn, how to use watermarks to configure event-time window behavior

Watermarks

- When an operator receives a watermark with time T, it can assume that no further events with timestamp less than T will be received.
- Watermarks are essential for both
  - event-time windows and
  - operators handling out-of-order events
- Watermarks provide a configurable tradeoff between
  - results confidence and
  - Latency

Watermarks

- *Eager* watermark ensure low latency & lower confidence
  - late events might arrive after the watermark have to handle with some code
- Verses, if watermarks are too relaxed provides:
  - High confidence, but
  - Might unnecessarily increase processing latency
- In many real-world applications:
  - I. The system does not enough knowledge to perfectly determine watermark
  - II. In mobile: impossible to know how long a user might remain disconnected
  - III. simply relying on watermarks might not always be a good idea
- Solution:
  - Provide some mechanism to deal with events that might arrive after the watermark
  - Depending on the application requirements:
    - Ignore such events
    - Log them
    - Use them to correct previous results

Processing Time Versus Event Time

- Why processing time if event time solves all our problems?!
- Processing time can be useful in some cases, when:
  - lowest latency is required
    - Since late events and out-of-order events do not take into consideration
    - While a window simply needs to buffer up events and immediately trigger computation once the specified time length is reached.
    - Thus, for applications where speed is more important than accuracy, processing time comes in handy
  - Need to periodically report results in real time, independently of their accuracy
    - Ex: real-time monitoring dashboard

State and Consistency Models

- State is ubiquitous in data processing
- It is required by any nontrivial computation.
- To produce a result, a function accumulates state over a period of time or number of events (e.g., to compute an aggregation or detect a pattern).
- Stateful operators use both incoming events and internal state to compute their output
- Take a rolling aggregation operator that outputs the current sum of all the events it has seen so far.
- The operator keeps the current value of the sum as its internal state and updates it every time it receives a new event.
- Ex: consider an operator that raises an alert when it detects a “high temperature” event followed by a “smoke” event within 10 minutes.

State and Consistency Models

- To limit the state size, operators usually maintain some kind of summary or **synopsis** of the events seen so far.
- Stateful operators comes with a few implementation challenges:
  - State management
    - The system efficiently manage the state to make sure it is protected from concurrent updates
  - State partitioning
    - Parallelization gets complicated (results depend on both the state and incoming events)
    - Partition the state by a key and manage the state of each partition independently
  - State recovery
    - The third and biggest challenge that comes with stateful operators is ensuring that state can be recovered, and results will be correct even in the presence of failures

Task Failures

- Operator state in streaming jobs is very valuable and should be guarded against failures.
- If state gets lost during a failure, results will be incorrect after recovery.
- Streaming jobs run for long periods of time, and thus state might be collected over several days or even months.
- Reprocessing all input to reproduce lost state in the case of failures would be both very expensive and time-consuming.
- Stream processing should continue in the case of
  - task failures, but also provide
  - correctness guarantees (result and operator state)

Task Failures

WHAT IS A TASK FAILURE?

- A task is a processing step:
  - Receives the event, storing it in a local buffer;
    - will the event get lost?
  - Possibly updates internal state;
    - will it update it again after it recovers?
  - Produces an output record.
    - will the output be deterministic?

Task Failures
Result Guarantees

- mean the consistency of the internal state of the stream processor
- What the application code sees as state value after recovering from a failure
- Guaranteeing the consistency of an application’s state is not the same a guaranteeing consistency of its output
- Result guarantees categories:
  - AT-MOST-ONCE
  - AT-LEAST-ONCE
  - EXACTL-ONCE (Apache Flink)
  - END-TO-END EXACTLY-ONCE

Task Failures

Result Guarantees

AT-MOST-ONCE or “No guarantee”

- when a task fails is to do nothing to recover lost state and replay lost events

Use cases:

- approximate results
- When the lowest latency possible

Task Failures

Result Guarantees

AT-LEAST-ONCE – “more than once”

- In most real-world applications, the expectation is that events should not get lost
- Duplicate processing might be acceptable if application correctness only depends on the completeness of information
- To ensure at-least-once result correctness:
    - Persistent event logs
        - Write all events to durable storage
    - Record acknowledgments
        - buffer until its processing has been acknowledged the event can be discarded

Task Failures

Result Guarantees

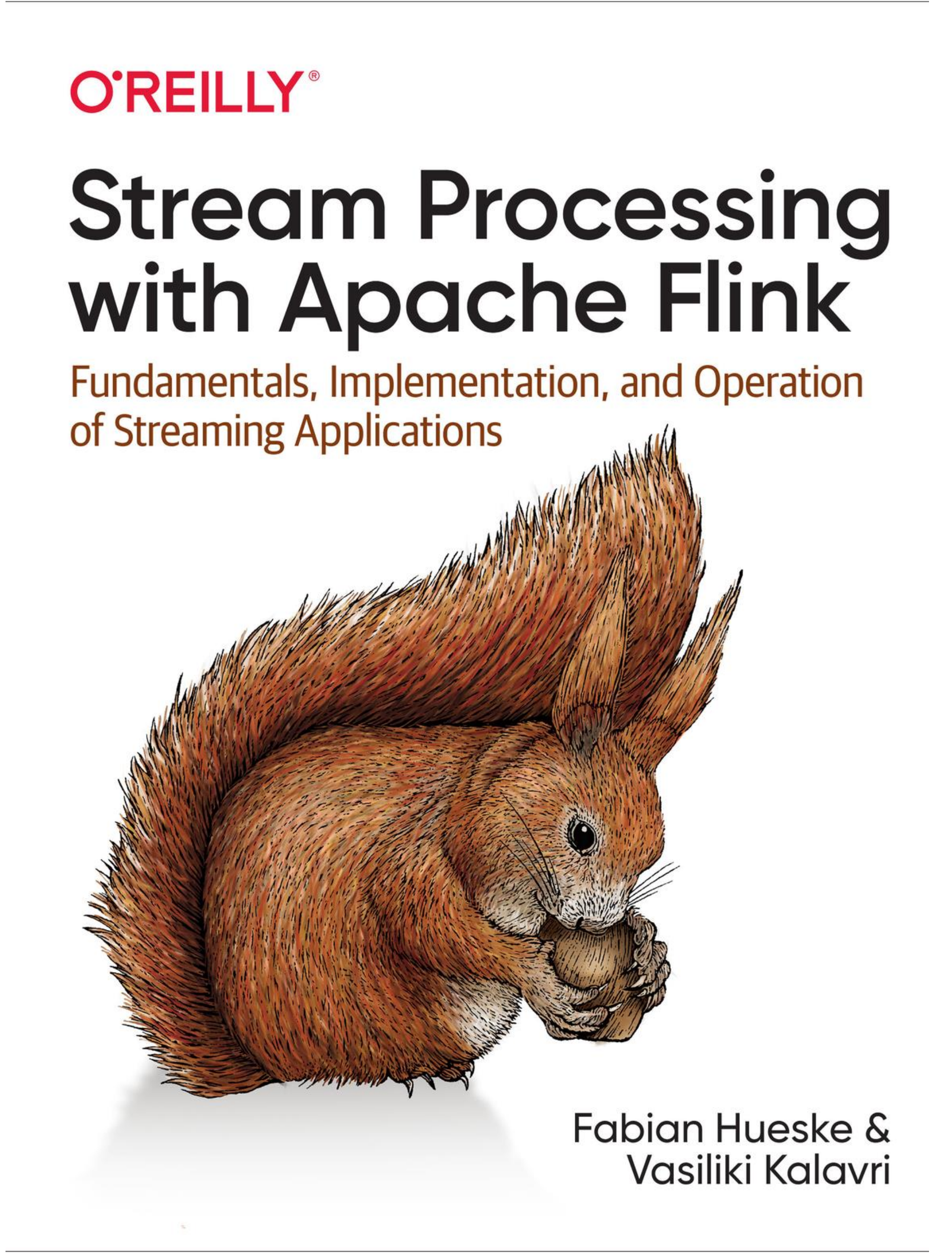
EXACTL-ONCE (Apache Flink)

- It is the strictest guarantee and hard to achieve
- Means, not only will there be no event loss
- Means, updates on the internal state will be applied exactly once for each event
- Means, Application will provide the correct result
- Like, a failure never happened
- Flink uses a lightweight snapshotting mechanism to achieve it

Task Failures
Result Guarantees

END-TO-END EXACTLY-ONCE

- It refer to result correctness across the whole data processing pipeline
- Each component provides its own guarantees
- The end-to-end guarantee of the complete pipeline would be the weakest of each of its components
- Note: Sometime get stronger semantics with weaker guarantees.
    - A common case is when a task performs idempotent operations, like maximum or minimum



O'REILLY

Stream Processing with Apache Flink

Fundamentals, Implementation, and Operation of Streaming Applications

Fabian Hueske & Vasiliki Kalavri

Chapter 4

Setting Up a Development Environment for Apache Flink

Required Software

- Flink applications can develop and execute on:
  - Linux (preferred by most Flink developers)
  - MacOS
  - Windows (WSL or Linux VM)
- Apache Maven 3.x
- An IDE (Recommended the IntelliJ IDEA)
- Java JDK 8 or higher (Go for JDK 11)
- Scala 2.12.18

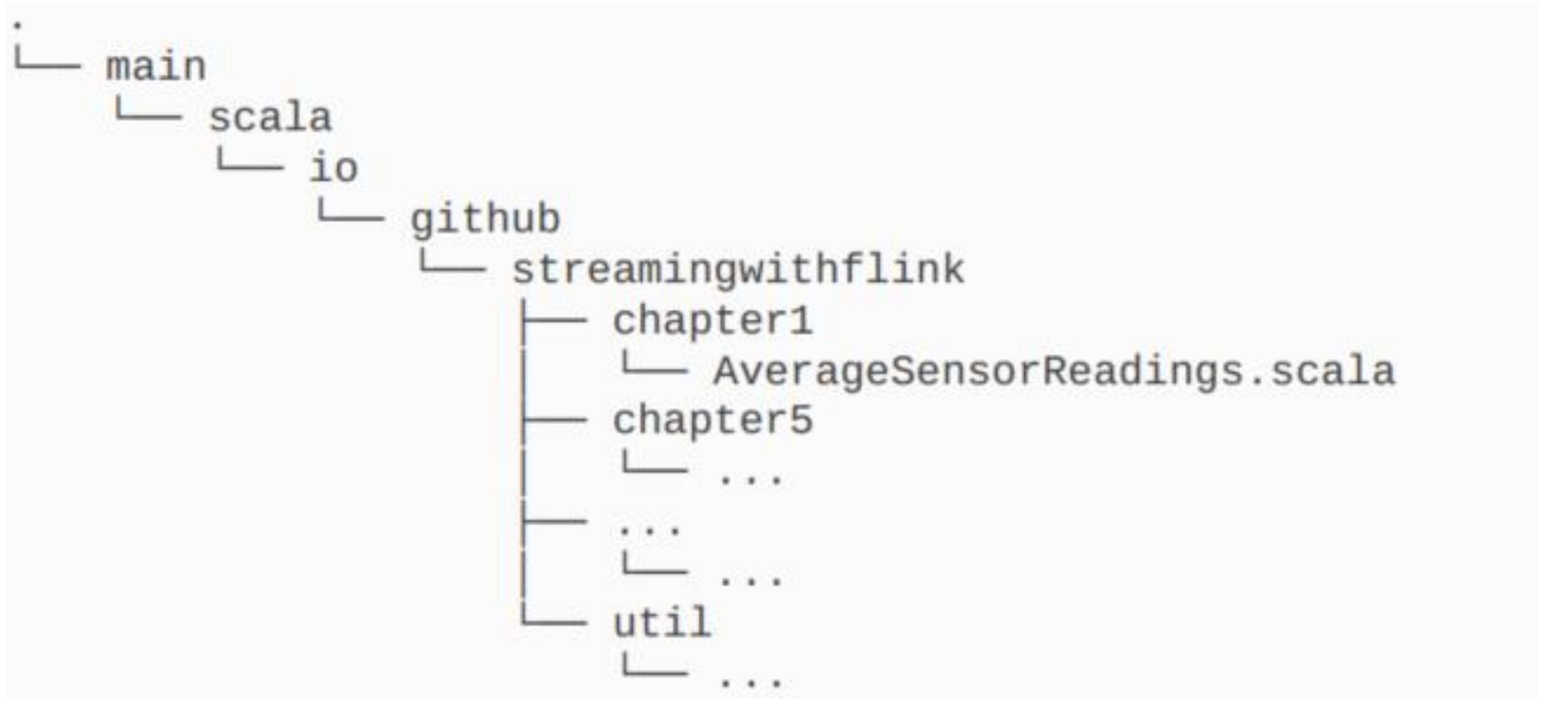
Run and Debug Flink Applications in an IDE

- Download examples from the Flink GitHub:
  - https://github.com/streaming-with-flink/examples-scala
  - https://github.com/streaming-with-flink/examples-java
- Clone the Scala version of Flink examples:
  - git clone https://github.com/streaming-with-flink/examples-scala

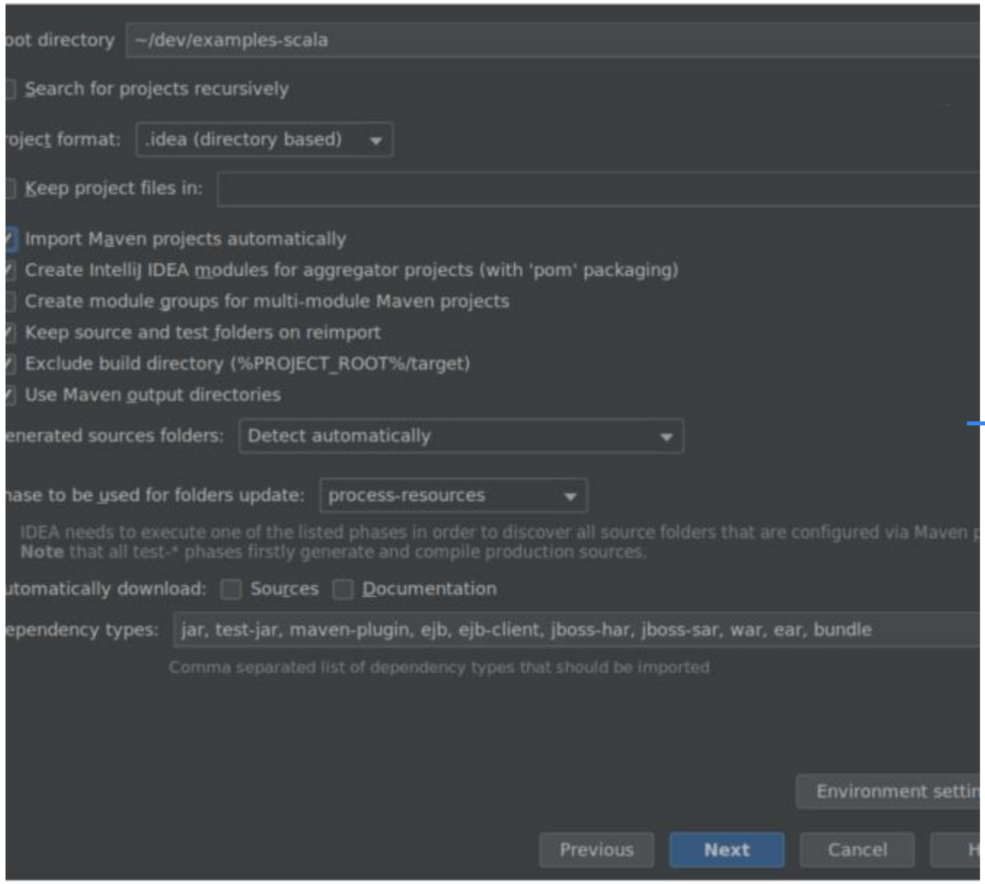
  Or
  - wget https://github.com/streaming-with-flink/examples-scala/archive/master.zip

Run and Debug Flink Applications in an IDE

- In the GitHub, the examples are provided as a Maven project grouped by chapter:



```
.
└── main
    └── scala
        └── io
            └── github
                └── streamingwithflink
                    ├── chapter1
                    │   └── AverageSensorReadings.scala
                    ├── chapter5
                    │   └── ...
                    ├── ...
                    │   └── ...
                    └── util
                        └── ...
```



oot directory ~/dev/examples-scala

☐ Search for projects recursively

oject format: .idea (directory based)

☐ Keep project files in:

✓ Import Maven projects automatically
✓ Create IntelliJ IDEA modules for aggregator projects (with 'pom' packaging)
☐ Create module groups for multi-module Maven projects
✓ Keep source and test folders on reimport
✓ Exclude build directory (%PROJECT_ROOT%/target)
✓ Use Maven output directories

enerated sources folders: Detect automatically

hase to be used for folders update: process-resources

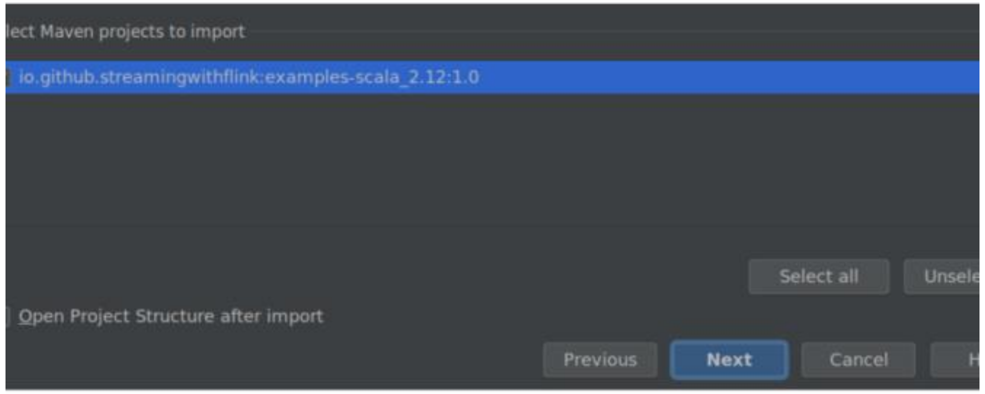
IDEA needs to execute one of the listed phases in order to discover all source folders that are configured via Maven p
Note that all test-* phases firstly generate and compile production sources.

utomatically download: ☐ Sources ☐ Documentation

ependency types: jar, test-jar, maven-plugin, ejb, ejb-client, jboss-har, jboss-sar, war, ear, bundle

Comma separated list of dependency types that should be imported

Import the book examples repository into IntelliJ



lect Maven projects to import

io.github.streamingwithflink:examples-scala_2.12:1.0

☐ Open Project Structure after import

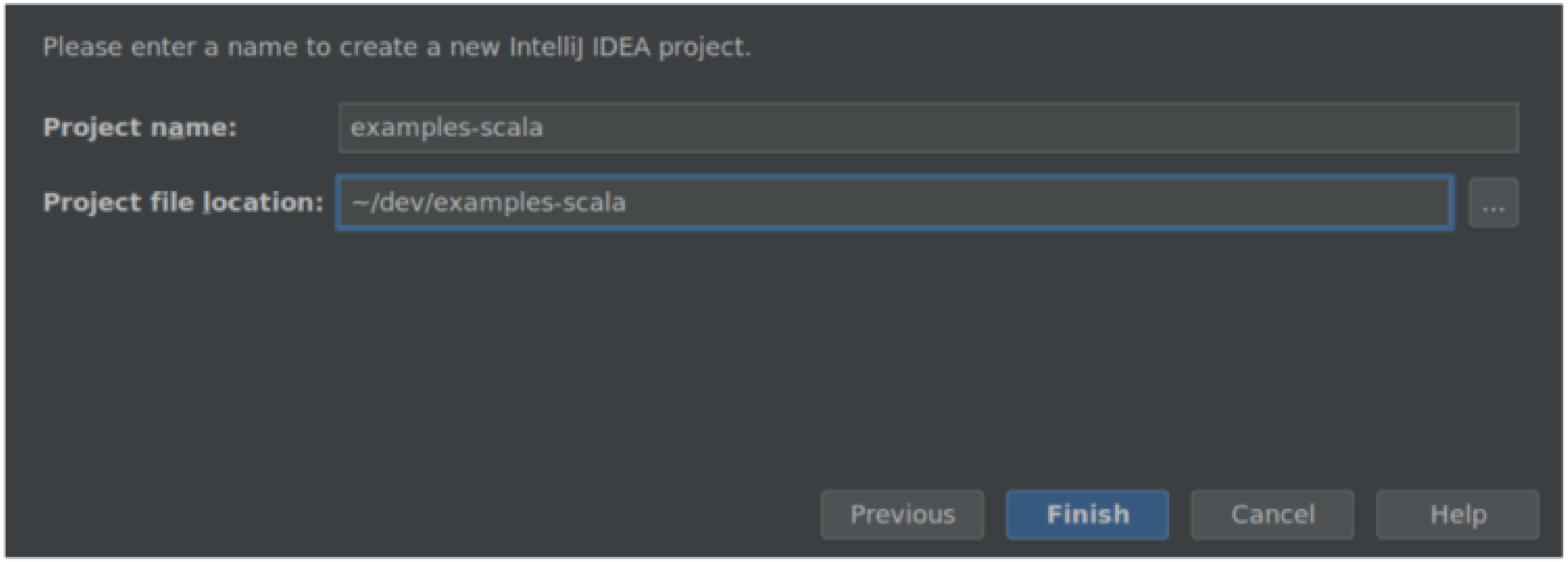
Select the Maven project to import

Run and Debug Flink Applications in an IDE

- Navigate to File -> New -> Project from Existing Sources,
- select the book examples folder “examples-scala”, and click OK.
- Make sure that “Import project from external model” and “Maven” are selected and
- click Next.
- A project import wizard will guide you though the next steps, such as
    - Selecting the Maven project to import (there should only be one),
    - selecting the SDK, and
    - naming the project

Run and Debug Flink Applications in an IDE

You should now be able to browse and inspect the code of the book examples



Please enter a name to create a new IntelliJ IDEA project.

Project name: examples-scala

Project file location: ~/dev/examples-scala

Give your project a name and click Finish

Run Flink Applications in an IDE

- Search for the *AverageSensorReadingsclass* and open it
- To start the application, run the ***main()*** method
- The output starts with a few log statements about the states that parallel operator tasks go through, such as SCHEDULING, DEPLOYING, and RUNNING. Output should be like this:

```
2> SensorReading(sensor_31,1515014051000,23.924656183848732)
4> SensorReading(sensor_32,1515014051000,4.118569049862492)
1> SensorReading(sensor_38,1515014051000,14.781835420242471)
3> SensorReading(sensor_34,1515014051000,23.871433252250583
```

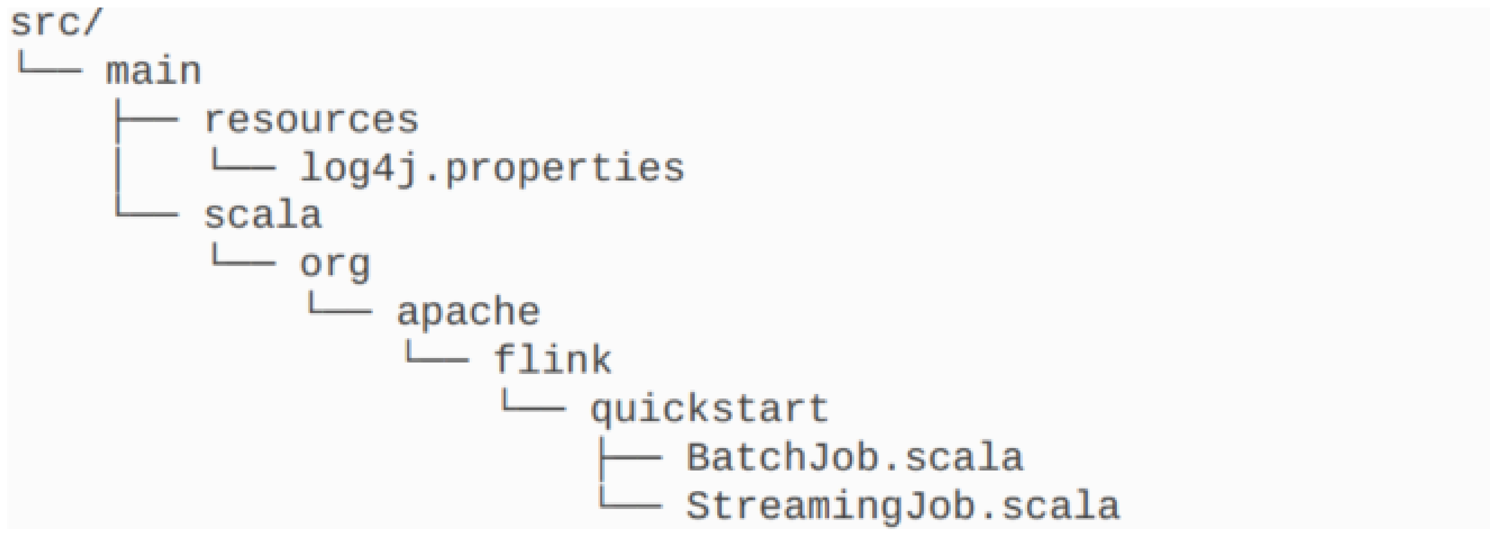
Bootstrap a Flink Maven Project

- Create a project from scratch
- Flink provides Maven archetypes to generate Maven projects for Java or Scala Flink applications.
- Open a terminal and run the following command to create a Flink Maven Quickstart Scala project as a starting point for your Flink application:

```
mvn archetype:generate                              \
      -DarchetypeGroupId=org.apache.flink           \
      -DarchetypeArtifactId=flink-quickstart-scala  \
      -DarchetypeVersion=1.7.1                      \
      -DgroupId=org.apache.flink.quickstart         \
      -DartifactId=flink-scala-project              \
      -Dversion=0.1                                 \
      -Dpackage=org.apache.flink.quickstart         \
      -DinteractiveMode=false
```

Bootstrap a Flink Maven Project

- This will generate Maven project for Flink 1.7.1 in a folder called *flink-scala-project*
- The generated folder contains a *src/* folder and a *pom.xml* file.
- The *src/* folder has the following structure:



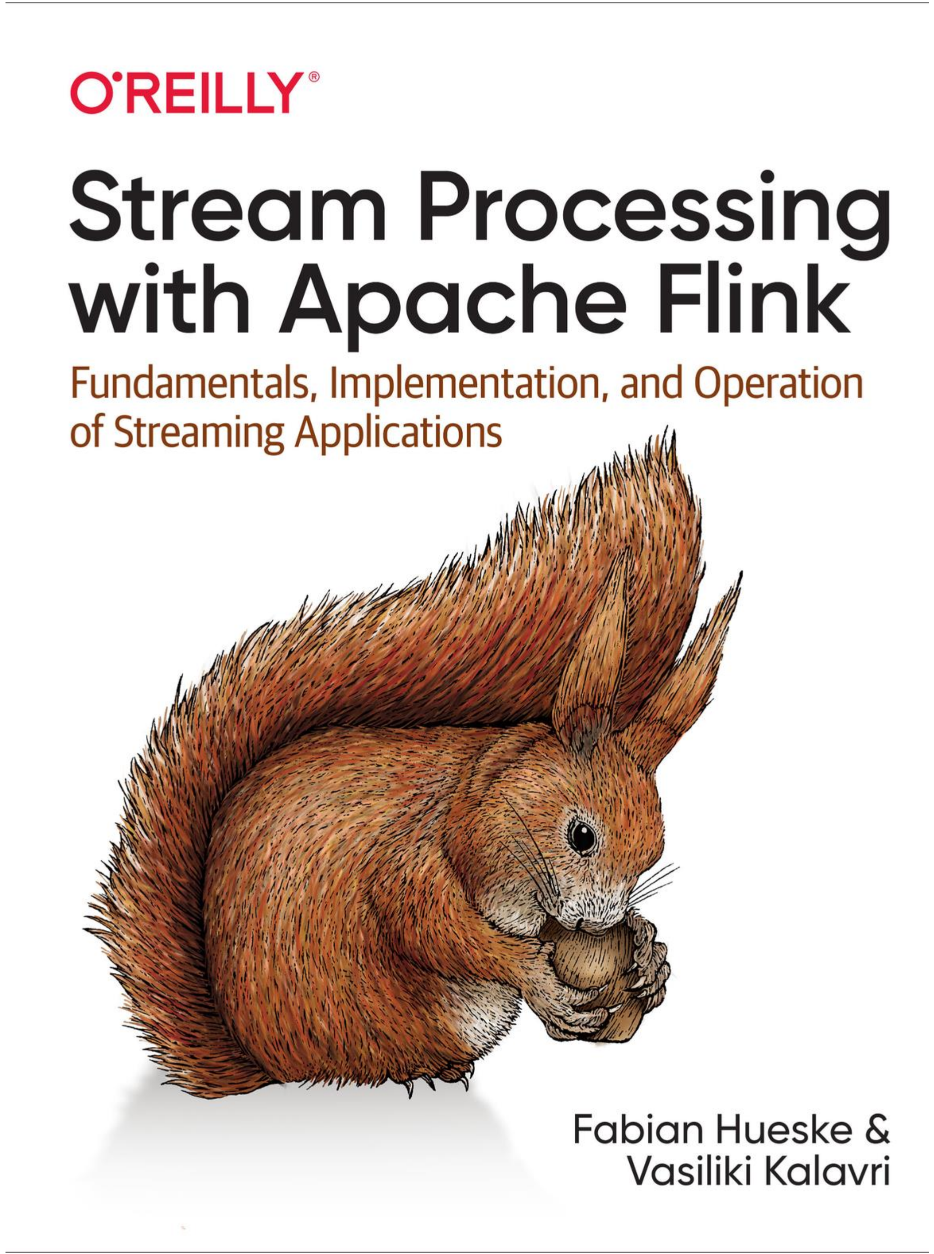
```
src/
└── main
    ├── resources
    │   └── log4j.properties
    └── scala
        └── org
            └── apache
                └── flink
                    └── quickstart
                        ├── BatchJob.scala
                        └── StreamingJob.scala
```

Bootstrap a Flink Maven Project

- you can execute the following command to build a JAR file:

mvn clean package -Pbuild-jar

- If the command completed successfully,
  - a new target folder created in the project folder.
  - The folder contains a file *flink-scalaproject-0.1.jar*, which is the JAR file of your Flink application.
  - The generated *pom.xml* file also contains instructions on how to add new dependencies to your project.


O'REILLY

Stream Processing with Apache Flink

Fundamentals, Implementation, and Operation of Streaming Applications

Fabian Hueske & Vasiliki Kalavri

Chapter 5

The DataStream API

Structure a typical Flink streaming application

1. Set up the execution environment.
2. Read one or more streams from data sources.
3. Apply streaming transformations to implement the application logic.
4. Optionally output the result to one or more data sinks.
5. Execute the program

Transformations - Basics

- Most stream transformations are based on user-defined functions.
- To encapsulate the user application logic and
- define how the elements of the input stream are transformed into the elements of the output stream.
- Below function implements a transformation-specific function interface

```
class MyMapFunction extends MapFunction[Int, Int] {
        override def map(value: Int): Int = value + 1
}
```

- Most function interfaces are designed as SAM (single abstract method)

Interfaces

- they can be implemented as Java 8 lambda functions
- The Scala DataStream API also has built-in support for lambda functions

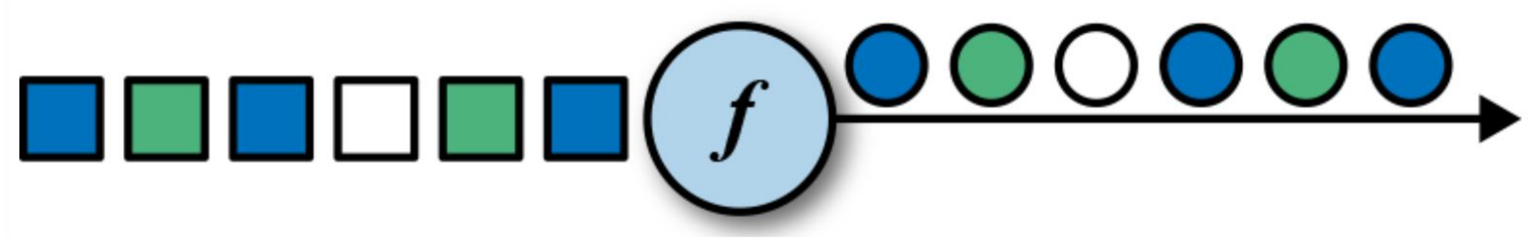
Transformations – Basics

Four categories of Transformations in DataStream API

1. Basic transformations are transformations on individual events.
2. *KeyedStream* transformations are transformations that are applied to events in the context of a key
3. Multistream transformations merge multiple streams into one stream or split one stream into multiple streams
4. Distribution transformations reorganize stream events

Transformations – Basics
Map Transformation

- Specified by calling the `DataStream.map()` transformation
- It produces a new `DataStream`
- It passes each incoming event to a user-defined mapper
- It returns exactly one output event



- It defines the `map()` method that transforms an input event into exactly one output event:
  - `// T: the type of input elements`
  - `// O: the type of output elements`
  - `MapFunction[T, O]`
  - `> map(T): O`

Transformations – Basics

Map Transformation

- The following is a simple mapper that extracts the first field (id) of each *SensorReading* in the input stream

```
val readings: DataStream[SensorReading] = ...
val sensorIds: DataStream[String] = readings.map(new MyMapFunction)
}

class MyMapFunction extends MapFunction[SensorReading,  String]      {
override def map(r: SensorReading): String = r.id
```

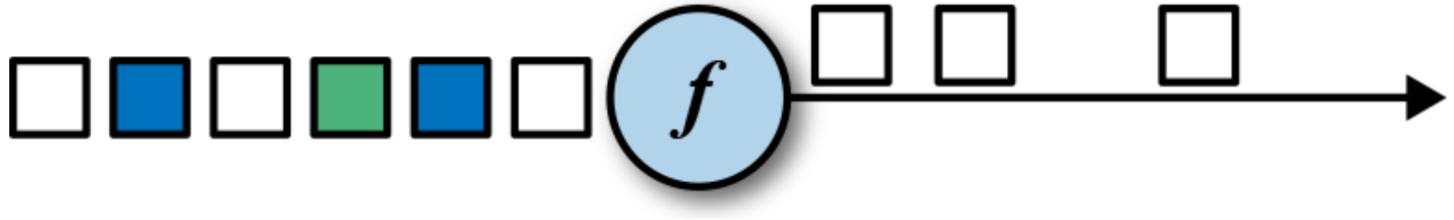
- Scala API or Java 8, the mapper can be expressed as lambda function

```
val readings: DataStream[SensorReading] =
...
val sensorIds: DataStream[String] =
readings.map(r => r.id)
```

Transformations – Basics

FILTER Transformation

- Drops or forwards events of a stream by evaluating a Boolean condition on each input event
  - True: preserves the input event and forwards it to the output
  - False: results in dropping the event.
- A filter transformation is specified by
  - Calling the **DataStream.filter()** method
  - Produces a new **DataStream** of the **same type** as the input DataStream
  - `// T: the type of elements`
  - `FilterFunction[T]`
  - `> filter(T): Boolean`



```
val sentences: DataStream[String] = ...
val words: DataStream[String] = sentences
.flatMap(id => id.split(" "))
```

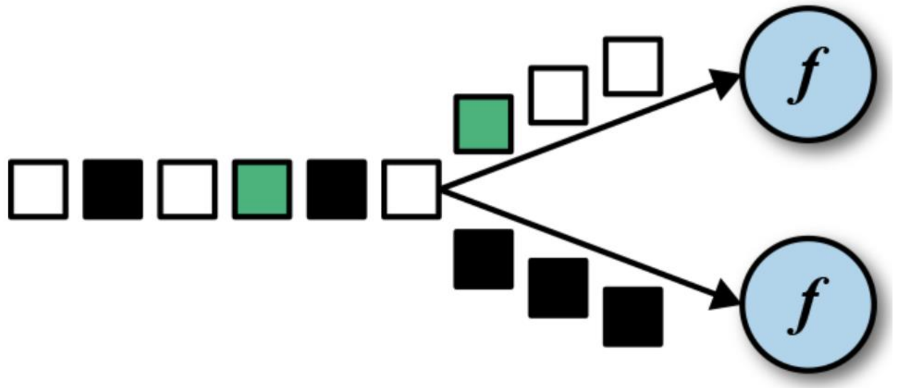
KeyedStream Transformations

- Groups of events that share a certain property together
- The DataStream API features the abstraction of a KeyedStream,
- which is a DataStream that has been logically partitioned into disjoint substreams of events that share the same key.
- Stateful transformations in the context of the currently processed event's key
  - all events with the same key access the same state and thus can be processed together
- If the key domain is continuously growing, you must clean up state for keys

KeyedStream Transformations
KEYBY

- The keyBy transformation converts a DataStream into a KeyedStream by specifying a key
- Based on the key, the events of the stream are assigned to partitions,
- so that all events with the same key are processed by the same task of the subsequent operator
- Events with different keys can be processed by the same task
- The following code declares the id field as the key of a stream of *SensorReading* records

```
val readings: DataStream[SensorReading]
= ...
val keyed: KeyedStream[SensorReading,
String] = readings.keyBy(r => r.id)
```



A keyBy operation that partitions events based on color

KeyedStream Transformations

ROLLING AGGREGATIONS Transformations

- Applied on a `KeyedStream` and produce a `DataStream` of aggregates,
  - sum, minimum, and maximum
- Keeps an aggregated value for every observed key
- Only use rolling aggregation on bounded key domains

```
val inputStream: DataStream[(Int, Int, Int)] = env.fromElements( (1, 2, 2),
(2, 3, 1), (2, 2, 4), (1, 5, 3))

val resultStream: DataStream[(Int, Int, Int)] = inputStream
    .keyBy(0) // key on first field of the tuple
    .sum(1) // sum the second field of the tuple in place
```

O/P: `(1,2,2)` followed by `(1,7,2)` and `(2,3,1)` followed by `(2,5,1)`

KeyedStream Transformations
REDUCE

- It's a generalization of the rolling aggregation
- Applies a ReduceFunction on a KeyedStream, which combines each incoming event with the current reduced value, and produces a DataStream
- It does not change the type of the stream
- The function can be specified with a class that implements the ReduceFunction interface. When its defines the reduce() method
- takes two input events and returns an event of the same type:
    - // T: the element type
    - ReduceFunction[T]
    - > reduce(T, T): T

```
val inputStream: DataStream[(String, List[String])] = env.fromElements( ("en",
List("tea")), ("fr", List("vin")), ("en", List("cake")))

val resultStream: DataStream[(String, List[String])] = inputStream .keyBy(0)

        .reduce((x, y) => (x._1, x._2 ::: y._2))
```

Multistream Transformations
UNION

- The DataStream.union() method merges two or more DataStreams of the same type and produces a new DataStream of the same type
- The events are merged in a FIFO fashion (no-ordering on events)
- Does not perform duplication elimination
- Every input event is emitted to the next operator

```
val parisStream: DataStream[SensorReading] = ...
val tokyoStream: DataStream[SensorReading] = ...
val rioStream: DataStream[SensorReading] = ...
val allCities: DataStream[SensorReading] = parisStream.union(tokyoStream, rioStream)
```

Multistream Transformations
CONNECT, COMAP, AND COFLATMAP

- Combining events of two streams is a very common requirement in stream processing.
- Ex: an application that monitors a forest area and outputs an alert whenever there is a high risk of fire
- The DataStream API provides the connect transformation to support such use cases
- The DataStream.connect() method receives a DataStream and returns a ConnectedStreams object, which represents the two connected streams:

```
// first stream
val first: DataStream[Int] = ...
// second stream
val second: DataStream[String] = ...
// connect streams
val connected: ConnectedStreams[Int, String] = first.connect(second)
```

- The ConnectedStreams object provides map() and flatMap() methods that expect a CoMapFunction and CoFlatMapFunction as argument respectively

Multistream Transformations
CONNECT, COMAP, AND COFLATMAP

- Both functions are typed on the types of the first and second input stream and on the type of the output stream and define two methods
- `map1()` and `flatMap1()` are called to process an event of the first input
- `map2()` and `flatMap2()` are invoked to process an event of the second input

```
// IN1: the type of the first input stream
// IN2: the type of the second input stream
// OUT: the type of the output elements
CoMapFunction[IN1, IN2, OUT]
> map1(IN1): OUT
> map2(IN2): OUT
// IN1: the type of the first input stream
// IN2: the type of the second input stream
// OUT: the type of the output elements
CoFlatMapFunction[IN1, IN2, OUT]
> flatMap1(IN1, Collector[OUT]): Unit
> flatMap2(IN2, Collector[OUT]): Unit
```

A FUNCTION CANNOT CHOOSE WHICH CONNECTEDSTREAMS TO READ

Multistream Transformations

CONNECT, COMAP, AND COFLATMAP

- In order to achieve deterministic transformations on ConnectedStreams, connect() can be combined with keyBy() or broadcast()

```
val one: DataStream[(Int, Long)] = ...
val two: DataStream[(Int, String)] = ...
// keyBy two connected streams
val keyedConnect1: ConnectedStreams[(Int, Long), (Int, String)] = one
.connect(two)
.keyBy(0, 0) // key both input streams on first attribute

// alternative: connect two keyed streams
val keyedConnect2: ConnectedStreams[(Int, Long), (Int, String)] = one
.keyBy(0)
.connect(two.keyBy(0)
```

Multistream Transformations

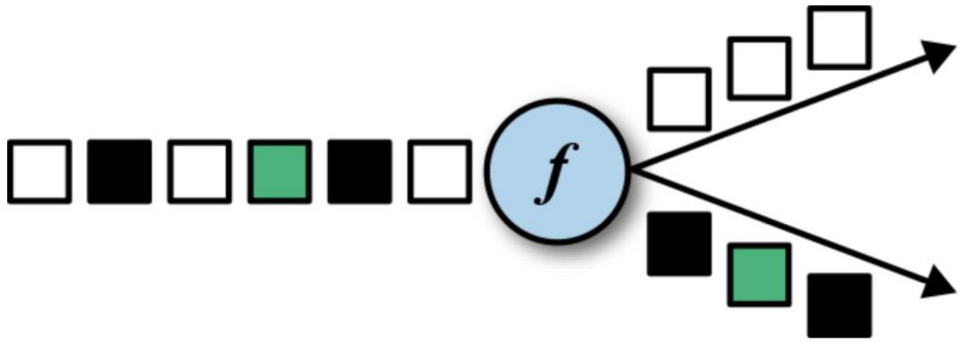
CONNECT, COMAP, AND COFLATMAP

- All events from both streams with the same key to the same operator instance
- Note that the keys of both streams should refer to the same class of entities
- with a broadcasted stream:

```
val first: DataStream[(Int, Long)] = ...
val second: DataStream[(Int, String)] = ...// connect streams with broadcast
val keyedConnect: ConnectedStreams[(Int, Long), (Int, String)] = first
// broadcast second input stream
.connect(second.broadcast())
```

Multistream Transformations

SPLIT AND SELECT



- Split is the inverse transformation to the union transformation
- Each incoming event can be routed to zero, one, or more output streams
- The DataStream.split() method returns a SplitStream, which provides a select() method to select one or more streams from the SplitStream by specifying the output names

```
val inputStream: DataStream[(Int, String)] = ...
val splitted: SplitStream[(Int, String)] = inputStream
.split(t => if (t._1 > 1000) Seq("large") else Seq("small"))
val large: DataStream[(Int, String)] = splitted.select("large")
val small: DataStream[(Int, String)] = splitted.select("small")
val all: DataStream[(Int, String)] = splitted.select("small", "large")
```

Distribution Transformations

- define how events are assigned to tasks
- Sometimes partitioning strategies at the application level or define custom partitioners would be necessary
- There are six categories of distribution:
    - Random
    - Round-Robin
    - Rescale
    - Broadcast
    - Global
    - Custom

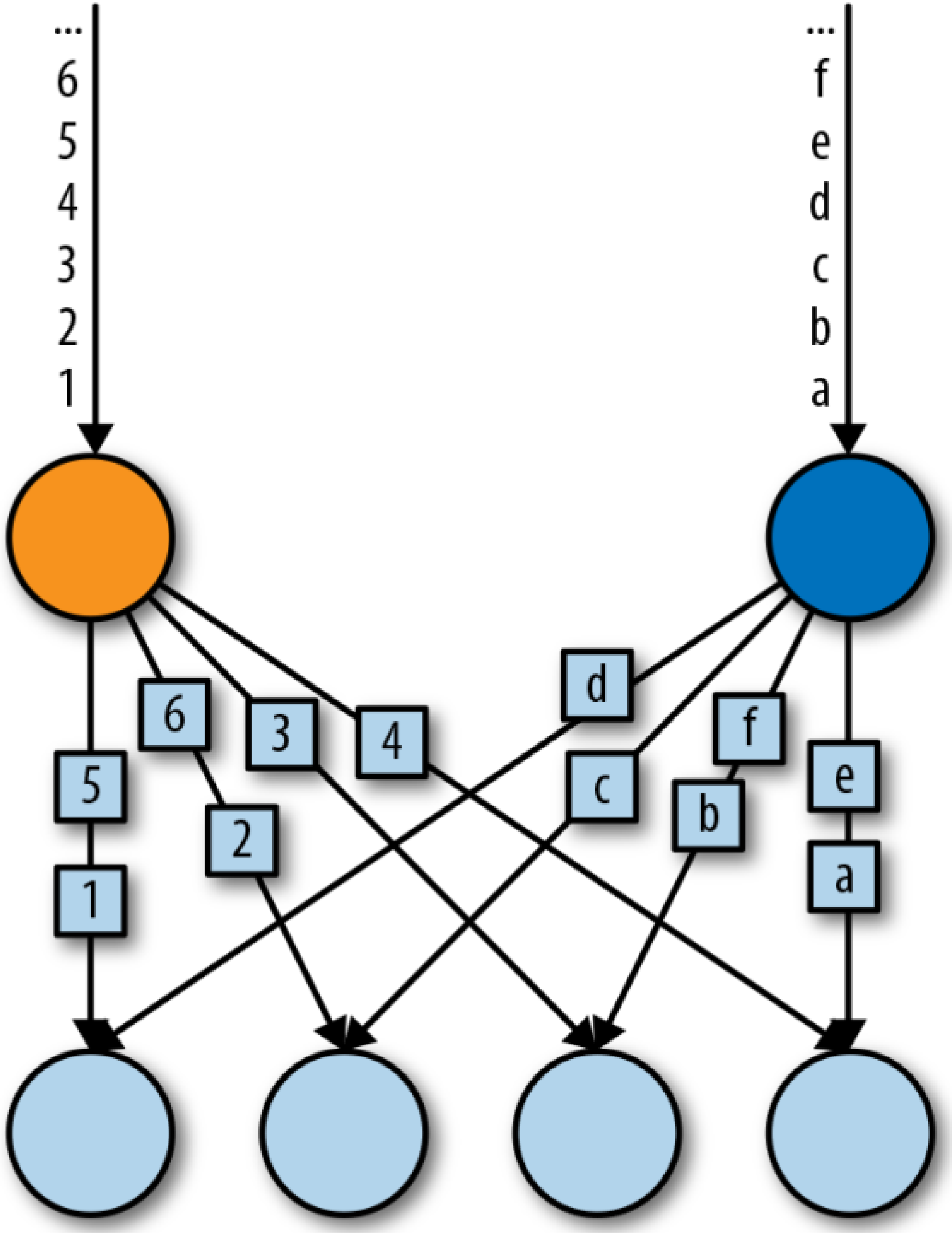
Distribution Transformations

Random data exchange strategy

- Implemented by the DataStream.shuffle() method
- The method distributes records randomly according to a uniform distribution to the parallel tasks of the following operator

Distribution Transformations
Round-Robin exchange strategy

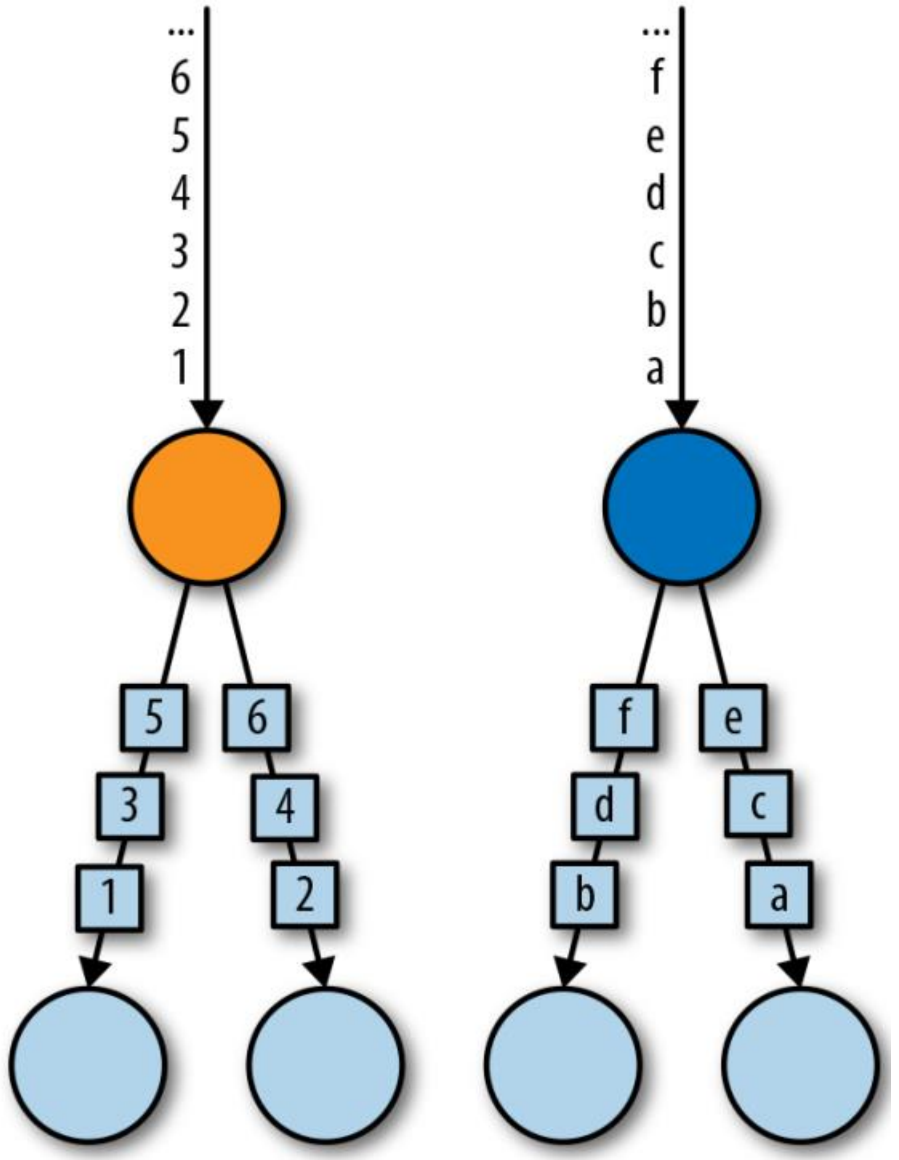
- The rebalance() method partitions the input stream
- The events are evenly distributed to successor tasks in a round-robin fashion



(a) Round-robin (rebalance)

Distribution Transformations
Rescale

- The rescale() method also distributes events in a round-robin fashion, but only to a subset of successor tasks.
- In essence, the rescale partitioning strategy offers a way to perform a lightweight load rebalance when the number of sender and receiver tasks is not the same.
- The rescale transformation is more efficient if the number of receiver tasks is a multitude of the number of sender tasks or vice versa.
- The fundamental difference between rebalance() and rescale() lies in the way task connections are formed. While rebalance() will create communication channels between all sending tasks to all receiving tasks, rescale() will only create channels from each task to some of the tasks of the downstream operator.



(b) Rescale

Distribution Transformations
Broadcast

- The broadcast() method replicates the input data stream
- so that all events are sent to all parallel tasks of the downstream operator.

Distribution Transformations
Global

- The global() method sends all events of the input data stream to the first
- parallel task of the downstream operator
- This partitioning strategy must be used with care, as routing all events to the same task might impact application performance

Distribution Transformations
Custom

- Predefined partitioning strategies is suitable so far?
- define your own by using the partitionCustom() method.
- This method receives a Partitioner object that implements the partitioning logic and
- the field or key position on which the stream is to be partitioned.
- The following example partitions a stream of integers so that all negative numbers are sent to the first task and all other numbers are sent to a random task

```
val numbers: DataStream[(Int)] = ...
numbers.partitionCustom(myPartitioner, 0)
object myPartitioner extends Partitioner[Int] {
        val r = scala.util.Random
        override def partition(key: Int, numPartitions: Int): Int = {
        if (key < 0) 0 else r.nextInt(numPartitions)
        }
}
```