Flink Training for Real-Time Data Engineering

The Flink DataStream API for Python

About Instructor

- About Instructor … *<text size should be 16 and style should be Trebuchet MS>*

NAME

Datacouch Instructor

CREATING INFINITE POSSIBILITIES

AGENDA

- Deep dive into the core API for building Flink applications.
- Sources: Reading data from streams (e.g., Kafka, Kinesis).
- Transformations: map, filter, keyBy, window, and other essential operations.
- Sinks: Writing data to destinations.

DATACOUCH
CREATING INFINITE POSSIBILITIES

DSAI
Data Science and AI Academy

Piramal Finance

Flink DataStream API for Python

PyFlink provides the **DataStream API** for building:

- **Parallel, distributed, real-time** data applications.

Runs on top of Flink's **streaming engine.**

Enables Python developers to leverage Flink's power for **stream & batch** jobs.

How It Works

Applications construct a **dataflow pipeline**:

1. **Sources** – Read data from streams (Kafka, files, sockets, etc.).
2. **Transformations** – Apply operations (map, filter, keyBy, window, reduce).
3. **Sinks –** Output results to storage, databases, or dashboards.

Dataflows are represented as **directed acyclic graphs (DAGs)**.

StreamExecutionEnvironment

Every PyFlink program starts with a **StreamExecutionEnvironment.**

Responsibilities:

- Manage **job lifecycle.**
- Define **parallelism & resources.**
- Trigger **execution** of the dataflow.

Example:

```
from pyflink.datastream import StreamExecutionEnvironment

env = StreamExecutionEnvironment.get_execution_environment()
```

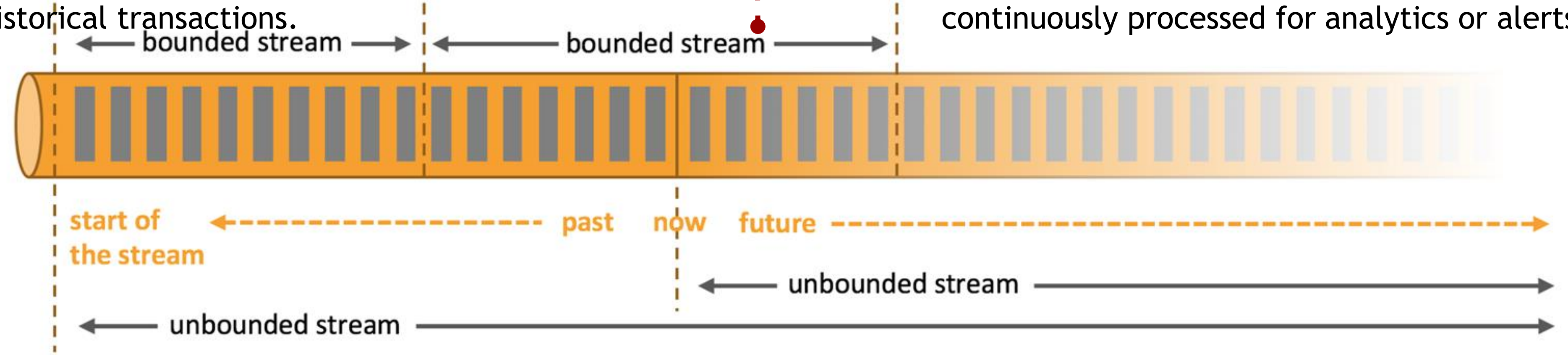
Bounded vs. Unbounded Streams

Bounded Streams (Batch)

- Represent a fixed dataset with a defined start and end.
- Data processing runs over the entire dataset and then terminates.
- Equivalent to traditional batch processing where all data is available before computation starts.
- Bounded streams can be sorted and processed deterministically.
- Example use case: Processing a static file of historical transactions.

Unbounded Streams (Streaming)

- Represent infinite or continuous streams of data with a defined start but no predefined end.
- Data is generated continuously (e.g., user clicks, sensor events), requiring ongoing, incremental processing.
- Real-time event processing happens as soon as data arrives without waiting for all data.
- Processing requires special handling of time and ordering.
- Example use case: User activity tracking on a website, continuously processed for analytics or alerts.



DATACOUCH
CREATING INFINITE POSSIBILITIES
DSAI
Data Science and AI Academy
Piramal Finance

What is StreamExecutionEnvironment?

Entry point for every Flink application - the starting point to define and run streaming jobs.

Manages the **entire job lifecycle**: submission, execution, and monitoring.

Supports multiple **execution modes**:

- Local (development & testing)
- Cluster (production)
- Cloud environments

Provides configuration for **parallelism, checkpoints, buffer timeouts, and state management.**
Enables building **dataflow pipelines** before triggering execution.

Sources → Transformations → Sinks

Creating the Execution Environment

```
from pyflink.datastream import StreamExecutionEnvironment
env = StreamExecutionEnvironment.get_execution_environment()
```

Entry point for all PyFlink streaming jobs.

Automatically detects context and creates:

- **LocalStreamEnvironment** → Local/standalone JVM.
- **RemoteStreamEnvironment** → Flink cluster jobs.
- **Cloud/Managed environments** → Integrated deployments.

Configuring the Environment

Access job configuration:

```
execution_config = env.get_config()
execution_config.set_parallelism(4)
```

Key settings:

- Parallelism (task concurrency).
- Restart strategies (fault tolerance).
- Checkpointing (intervals & timeouts).
- Python-specific configs (batch size, UDF execution).

Advanced parameters:

- **python.fn-execution.bundle.size** → batch size for UDFs.
- **python.operator-chaining.enabled** → enable/disable chaining.
- Closure cleaner levels, memory tuning, metrics.

Managing Parallelism

Managing Parallelism in Apache Flink controls how many parallel instances (tasks) of an operator, source, or sink run across the cluster nodes. This enables scalable and efficient processing of streaming data.

Key Points on Parallelism

- **Parallelism** = number of concurrent tasks running for a Flink job or operator
- Can be set:
  - **Globally** → at the execution environment level
  - **Specifically** → per operator, source, or sink
- Ensures workload is **distributed** across available cluster resources

Example in PyFlink

```python
from pyflink.datastream import StreamExecutionEnvironment

env = StreamExecutionEnvironment.get_execution_environment()

# Set default parallelism for the entire job
env.set_parallelism(4)
```

Execution Lifecycle

1. Define Environment

- Create **StreamExecutionEnvironment**
- Abstracts target: Local JVM | Cluster | Cloud

2. Add Sources

- Ingest from **Kafka, files, sockets, APIs**

3. Apply Transformations

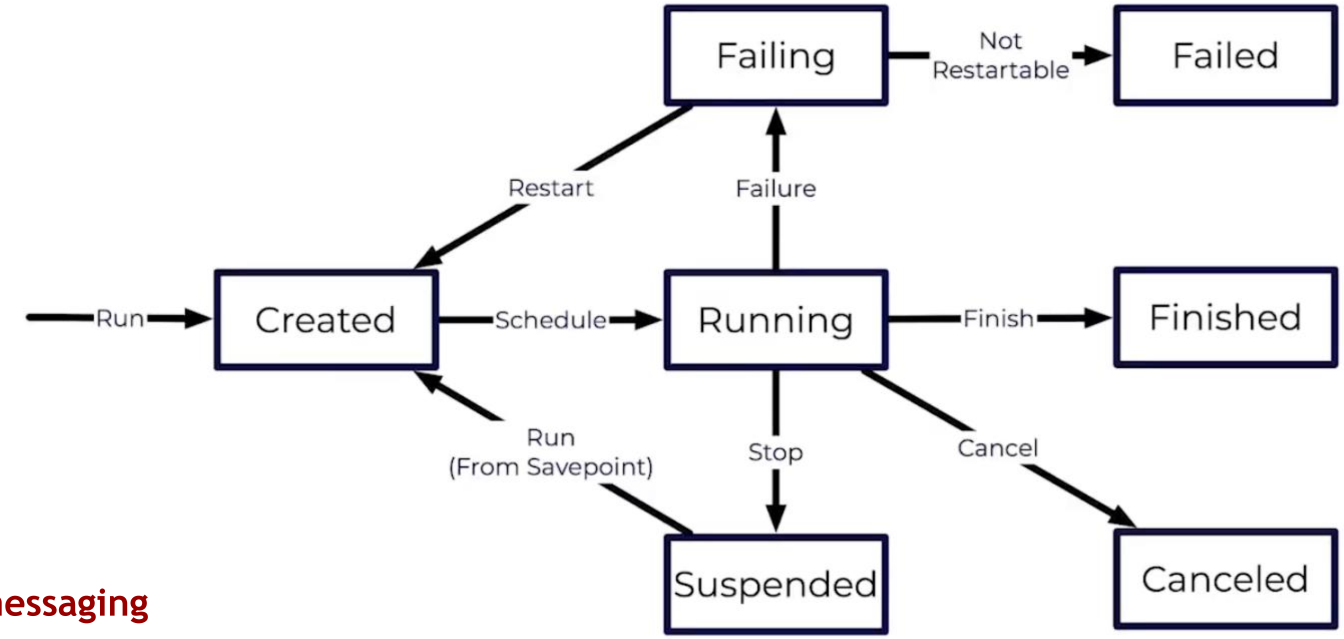
- Chain operations:
    - **map, filter, keyBy, window, reduce**

4. Add Sinks

- Output to **databases, storage, dashboards, messaging**

5. Trigger Execution

- env.execute("job_name") → submits job
- Builds **JobGraph** for scheduling



What are Sources?

- **Entry points** for data into a Flink job.
- Define **where and how** data enters the streaming pipeline.
- Can be:
    - Simple **collections** (testing/demo).
    - **Files** (CSV, JSON, etc.).
    - **Network sockets** (real-time feeds).
    - **Streaming platforms** (Kafka, Kinesis, Pulsar).
- Choice of source impacts **latency, throughput, and reliability**.

Built-in Sources in Flink

Collections – Create streams from in-memory data.

```
env.from_collection([1, 2, 3])
```

- **Files** – Read static or streaming data from filesystems (local or DFS).
- **Sockets** – Ingest data via TCP socket for quick prototyping.

Usage:

- Best for **demos, testing, or light workloads.**
- Simple and fast to set up.

External Sources in Flink

- **Kafka** – Leading event-streaming platform with partitions, durability, and exactly-once guarantees.
- **Kinesis** – AWS-managed service, elastic scaling, cloud-native integration.
- **Pulsar** – Scalable pub/sub with multi-tenancy and geo-replication.
- **Distributed Filesystems (HDFS, S3, etc.)** – Persistent storage for logs, batch, or checkpoints.

Why Use Them?

- High **throughput & reliability.**
- **Partitioning** for parallelism.
- **Exactly-once semantics** with Flink connectors.

Example: Kafka Source

Python example using FlinkKafkaConsumer:

```
from pyflink.datastream.connectors import FlinkKafkaConsumer
from pyflink.common.serialization import SimpleStringSchema

props = {'bootstrap.servers': 'localhost:9092', 'group.id': 'flink_group'}

ds = env.add_source(FlinkKafkaConsumer(
    topics='topic’,
    deserialization_schema=SimpleStringSchema(),
    properties=props
))
```

- This sets up a reliable and parallel Kafka source stream ingestion.
- Supports checkpointing and offset management internally.

Custom Sources

When to Use

- Specialized needs beyond built-in connectors
- Examples: **APIs, sensors, IoT devices, proprietary protocols**

How to Implement

- Extend **SourceFunction**
- Implement required methods:
  - **start** → initialize & fetch data
  - **emit** → push data into the stream
  - **cancel** → stop gracefully

Key Benefit

- Flexibility to integrate with **almost any data provider**

What are Transformations?

- **Operations applied to streams** that build the **dataflow DAG** in Flink.
- Enable developers to **modify, filter, group, and aggregate** data in real time.
- Represent the core logic of stream processing applications.

Key Types of Transformations

- **map** – Element-wise transformation.
- **filter** – Remove unwanted events.
- **keyBy** – Partition stream by key.
- **reduce** – Aggregate values by key.
- **window** – Group data into time/count windows.
- **flatMap** – Split elements into multiple outputs.
- **process** – Low-level control with access to state and timers.

Core Transformations with Examples

Map - Apply a function to each element

```
ds = ds.map(lambda x: x * 2)
```

Filter - Keep elements that satisfy a condition

```
ds = ds.filter(lambda x: x > 10)
```

KeyBy - Partition stream by key for grouped operations

```
ds = ds.key_by(lambda x: x.id)
```

Reduce - Aggregate values per key

```
ds = ds.key_by(lambda x: x.id) \
       .reduce(lambda a, b: a + b)
```

Advanced Control & Optimization in Flink

ProcessFunction

- Provides **fine-grained event processing.**
- Direct access to **state and timers.**
- Useful for **complex event-driven logic** (e.g., alerts, pattern detection).

Operator Chaining

- **Combines multiple operators** into a single task.
- Reduces s**cheduling & communication overhead.**
- Improves **runtime efficiency** of pipelines.

What are Sinks?

Define the destination of processed data after transformations.

- Represent the **end of a dataflow pipeline** in Flink.
- Common examples:
    a. **Files** (CSV, JSON, Parquet)
    b. **Databases** (JDBC, NoSQL)
    c. **Message queues** (Kafka, Pulsar)
    d. **Dashboards / external services**

Key Point:
Sinks make Flink results **actionable** by delivering them to storage, systems, or real-time consumers.

Built-in Sinks in Flink

Print Sink

- Outputs results directly to the console.
- Ideal for **development, testing, and debugging.**

File Sink

- Writes results to **local or distributed filesystems.**
- Supports formats like **CSV, JSON, Parquet.**
- Useful for persisting results in simple workflows.

Note:
Built-in sinks are lightweight and best suited for **demos, prototypes, or small-scale jobs.**

External Sinks in Flink

- **Kafka** – High-throughput, fault-tolerant event streaming.
- **JDBC** – Write results to **relational databases.**
- **Elasticsearch** – Power search & analytics use cases.
- **Others** – AWS S3, Pulsar, custom sinks via connectors.

Why External Sinks?

- Enable **scalable, production-grade pipelines.**
- Support **exactly-once delivery semantics** with Flink’s checkpointing.
- Integrate seamlessly into **enterprise data ecosystems.**

Example: Kafka Sink in PyFlink

```
from pyflink.datastream.connectors import FlinkKafkaProducer
from pyflink.common.serialization import SimpleStringSchema

props = {'bootstrap.servers': 'localhost:9092'}

ds.add_sink(FlinkKafkaProducer(
    topic='topic',
    serialization_schema=SimpleStringSchema(),
    producer_config=props
))
```

Key Points

- FlinkKafkaProducer connects Flink pipelines to Kafka topics.
- Ensures high-throughput, fault-tolerant event delivery.
- Supports serialization schemas (e.g., JSON, Avro, String).
- Enables real-time streaming pipelines with external systems.

Putting It Together

- A Flink job is defined as:
  Sources → Transformations → Sinks
- This forms a **Directed Acyclic Graph (DAG)** representing the dataflow.
- The DAG is executed in parallel across the cluster.
- Ensures **scalable, fault-tolerant, real-time processing.**

Example PyFlink Pipeline:

```
ds = env.from_collection([1, 2, 3, 4]) \
      .map(lambda x: x * 2) \
      .filter(lambda x: x > 5)

ds.print()

env.execute("example_job")
```

Summary & Key Takeaways

- **DataStream API** → Flexible, powerful, and scalable for real-time applications.
- **Core building blocks** →
  a. **Sources** (data ingestion)
  b. **Transformations** (processing logic)
  c. **Sinks** (outputs/results)
- **PyFlink** → Pythonic access to Flink's **runtime, connectors, and ecosystem.**
- Enables building **production-ready streaming pipelines** with **state, fault tolerance, and parallelism.**

ANY SUGGESTIONS?

DATACOUCH
CREATING INFINITE POSSIBILITIES

DSAI
Data Science and AI Academy

Piramal Finance

THANK YOU