Apache Flink

Overview of Stream Processing

Stream processing is a paradigm for processing data as a continuous flow, rather than in batches. This approach is ideal for applications that require real-time insights and low latency. Unlike traditional batch processing, which involves collecting data into batches before processing, stream processing handles data as it arrives, offering immediate results.

Key characteristics of stream processing:

- ***Continuous data flow:*** Data is processed as it arrives, without the need for batching.
- ***Low latency***: Results are generated quickly, often within milliseconds.
- ***High throughput***: Can handle large volumes of data efficiently.
- ***Scalability***: Can handle increasing data volumes by adding more resources.

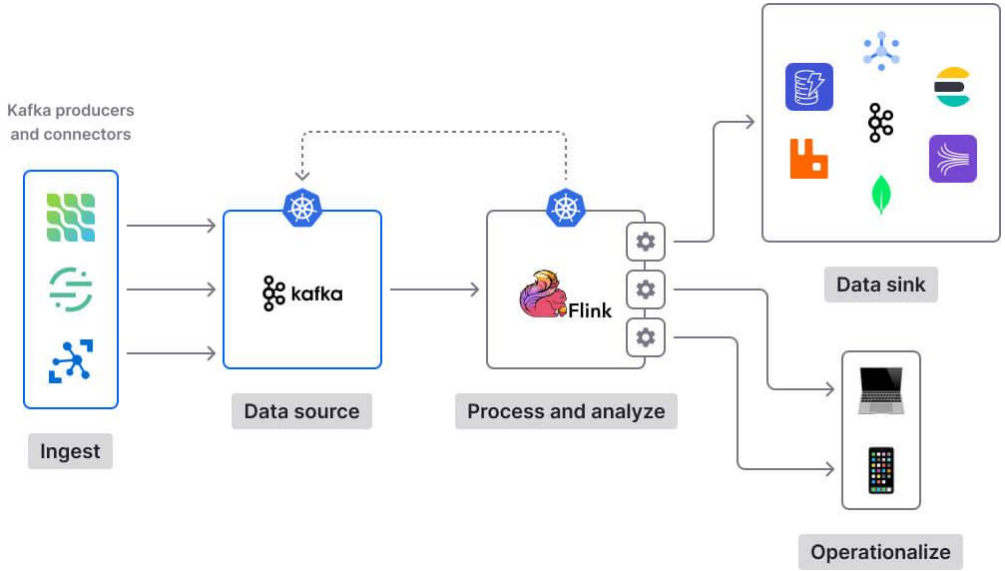
Introduction to Apache Flink

Apache Flink is an open-source platform for distributed stream processing. It's designed to execute applications in a distributed manner, providing low latency and high throughput. Flink can process both bounded and unbounded data streams, making it versatile for various applications.

Key Features of Apache Flink:

- ***Event-Time Processing*** - Supports out-of-order data with watermarks for accurate time-based operations.
- ***Stateful Stream Processing:***
    - Provides state management for advanced use cases like fraud detection or sessionization.
    - Offers exactly-once semantics for state consistency.

- ***Fault Tolerance*** - Uses checkpointing and distributed snapshots for automatic recovery.
- ***Flexible Deployment:***
  - Supports on-premise, cloud, and Kubernetes environments.
  - Integrates with YARN, Kubernetes, Mesos, or standalone clusters.
- ***API Richness:***
  - High-level APIs (DataStream, Table API, SQL).
  - Supports multiple languages: Java, Scala, Python.
- ***Seamless Integration*** - Works well with Apache Kafka, HDFS, Cassandra, Elasticsearch, and other modern tools.



Apache Flink APIs

Grow Data Skills

Flink offers different levels of APIs for developing streaming/batch applications.

- ***DataStream API (Core API)*** - The DataStream API is the core API for working with streams. It's used to define transformations on streams, offering flexibility without going into low-level details. Provides access to stream transformations like filter, map, flatMap, etc.

```
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.datastream.connectors.kafka import FlinkKafkaConsumer, FlinkKafkaProducer
from pyflink.datastream.formats.json import JsonRowDeserializationSchema,
JsonRowSerializationSchema
from pyflink.common import Types

def datastream_api_with_kafka():
    # Initialize the Flink environment
    env = StreamExecutionEnvironment.get_execution_environment()
    env.set_parallelism(1)

    # Kafka Source
    source_type_info = Types.ROW([Types.INT(), Types.STRING(), Types.DOUBLE()])
    deserialization_schema = JsonRowDeserializationSchema.builder() \
        .type_info(source_type_info) \
        .build()

    kafka_consumer = FlinkKafkaConsumer(
        topics="source_topic",
        deserialization_schema=deserialization_schema,
        properties={
            "bootstrap.servers": "localhost:9092",
            "group.id": "flink-group"
        }
    )
```

Apache Flink APIs


```
    kafka_consumer.set_start_from_earliest()

    # Kafka Sink
    sink_type_info = Types.ROW([Types.INT(), Types.STRING(), Types.DOUBLE()])
    serialization_schema = JsonRowSerializationSchema.builder() \
        .with_type_info(sink_type_info) \
        .build()

    kafka_producer = FlinkKafkaProducer(
        topic="sink_topic",
        serialization_schema=serialization_schema,
        producer_config={"bootstrap.servers": "localhost:9092"}
    )

    # Data Pipeline
    input_stream = env.add_source(kafka_consumer)

    # Transform the data: filter records with `total_price > 100` and capitalize the `order_status`
    processed_stream = input_stream \
        .filter(lambda row: row[2] > 100) \
        .map(lambda row: (row[0], row[1].upper(), row[2] * 1.2),
             output_type=sink_type_info)

    # Sink the processed data back to Kafka
    processed_stream.add_sink(kafka_producer)

    # Execute the Flink job
    env.execute("DataStream API with Kafka")

if __name__ == "__main__":
    datastream_api_with_kafka()
```

Apache Flink APIs

Grow Data Skills

- ***Table API (Declarative DSL) -*** The Table API provides a higher-level abstraction compared to the DataStream API. It's a programmatic, SQL-like DSL for working with structured data in tables. Allows relational-style operations like select, filter, and group_by. Integrated with the SQL API for seamless querying.

```
from pyflink.table import EnvironmentSettings, TableEnvironment
from pyflink.table.expressions import col

env_settings = EnvironmentSettings.new_instance().in_streaming_mode().build()
table_env = TableEnvironment.create(env_settings)

# Define Kafka source table
table_env.execute_sql("""
    CREATE TABLE source_table (
        order_id INT,
        order_status STRING,
        total_price DOUBLE
    ) WITH (
        'connector' = 'kafka',
        'topic' = 'source-topic',
        'properties.bootstrap.servers' = 'localhost:9092',
        'format' = 'json'
    )
""")

# Transformation
source_table = table_env.from_path("source_table")
transformed_table = source_table.filter(col("total_price") > 100) \
    .select(
        col("order_id"),
        col("order_status").upper_case(),
        (col("total_price") * 1.2).alias("total_price")
    )

transformed_table.execute().print()
```

Apache Flink APIs

Grow Data Skills

- ***SQL API (High-Level Language)*** - The SQL API allows developers to define transformations using SQL queries. It's perfect for use cases where business analysts or engineers prefer working with SQL. Ideal for static queries.

```
from pyflink.table import EnvironmentSettings, TableEnvironment

env_settings = EnvironmentSettings.new_instance().in_streaming_mode().build()
table_env = TableEnvironment.create(env_settings)

table_env.execute_sql("""
    CREATE TABLE source_table (
        order_id INT,
        order_status STRING,
        total_price DOUBLE
    ) WITH (
        'connector' = 'kafka', 'topic' = 'source-topic', 'properties.bootstrap.servers' = 'localhost:9092', 'format' = 'json'
    )
""")

table_env.execute_sql("""
    CREATE TABLE sink_table (
        order_id INT,
        order_status STRING,
        total_price DOUBLE
    ) WITH (
        'connector' = 'kafka', 'topic' = 'sink-topic', 'properties.bootstrap.servers' = 'localhost:9092', 'format' = 'json'
    )
""")

# SQL query to transform and insert data
table_env.execute_sql("""
    INSERT INTO sink_table
    SELECT order_id, UPPER(order_status), total_price * 1.2
    FROM source_table WHERE total_price > 100
""")
```

Programs and Dataflows in Apache Flink

Grow Data Skills

A Flink program typically consists of the following components:

- ***Source***: Where the data comes from (e.g., Kafka, files, or custom sources)
- ***Transformations***: Operations applied to the data
- ***Sink***: Where the results are sent (e.g., databases, files, or message queues)

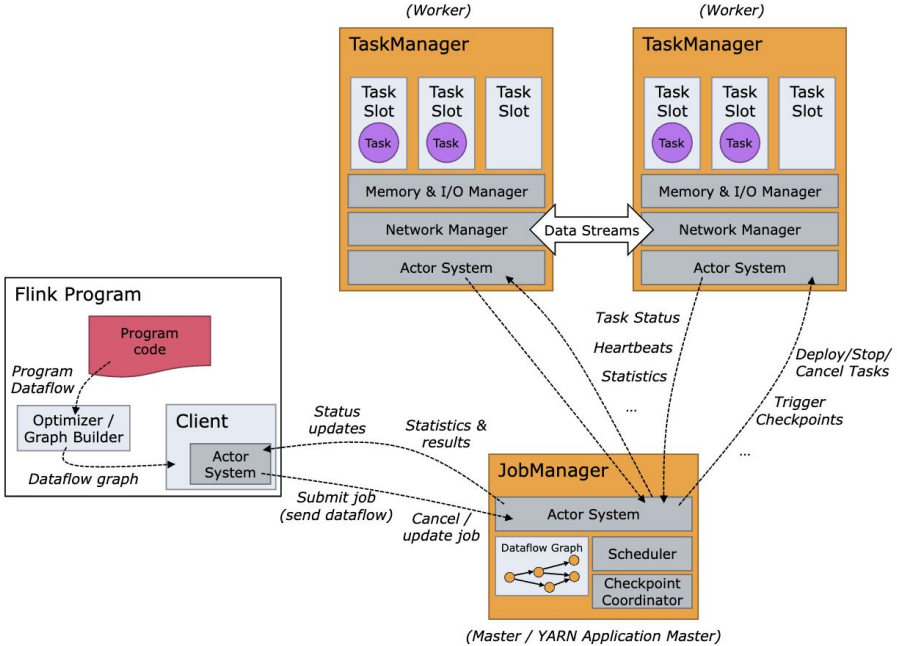
When a Flink program is executed, it is translated into a **dataflow graph**. This graph consists of:

- ***Streams***: Represent datasets in motion
- ***Operators***: Represent transformations on streams

Apache Flink Architecture

Grow Data Skills

Flink is a distributed system and requires effective allocation and management of compute resources in order to execute streaming applications. It integrates with all common cluster resource managers such as **Hadoop YARN** and **Kubernetes**, but can also be set up to run as a **standalone cluster** or even as a library.



The Flink runtime consists of two types of processes: a **JobManager** and one or more **TaskManagers**.

The Client is not part of the runtime and program execution, but is used to prepare and send a dataflow to the JobManager. After that, the client can disconnect (detached mode), or stay connected to receive progress reports (attached mode). The client runs either as part of the Java/Scala program that triggers the execution, or in the command line process **./bin/flink run ....**

Apache Flink Architecture

Grow Data Skills

Job Manager - The Job Manager is the central coordinator of a Flink cluster. Its primary responsibilities include:

- Scheduling tasks
- Coordinating checkpoints
- Coordinating failure recovery
- Managing the control flow of the job

The Job Manager consists of several components:

- ***ResourceManager:*** Responsible for resource de-/allocation and provisioning in a Flink cluster.
- ***Dispatcher***: Provides a REST interface to submit applications and starts a new JobMaster for each submitted job.
- ***JobMaster***: Coordinates the execution of a single job. Multiple JobMasters can run in a cluster, each responsible for a different job.

Task Managers - The Task Managers (also called workers) execute the tasks of a dataflow, and buffer and exchange the data streams.

There must always be at least one Task Manager. The smallest unit of resource scheduling in a Task Manager is a **task slot**. The number of task slots in a Task Manager indicates the number of **concurrent processing tasks**. Note that multiple operators may execute in a task slot.

Apache Flink Architecture

Grow Data Skills

Operators - An operator in Apache Flink is a basic building block that performs specific processing tasks, such as map, filter, join, or sink. These are the core components of a Flink pipeline.

```
stream = env.from_source(kafka_source)
mapped_stream = stream.map(lambda x: x.upper())
filtered_stream = mapped_stream.filter(lambda x: "ERROR" in x)
filtered_stream.sink_to(kafka_sink)
```

Operators here:

- ***Source***: Reads from Kafka (from_source).
- ***Map***: Converts data to uppercase (map).
- ***Filter***: Filters only messages containing "ERROR" (filter).
- ***Sink***: Writes back to Kafka (sink_to).

Tasks - A task is the runtime representation of an operator. Each operator instance runs as a task on a Task Manager. If an operator has a parallelism of 4, there will be 4 tasks for that operator.

Key Points:

- A task is bound to a single parallel instance of an operator.
- The number of tasks is determined by the parallelism.

Example (from above):

- If parallelism = 2, the map operator will have 2 tasks, and the same applies to the filter and sink.

Apache Flink Architecture

Grow Data Skills

Parallelism - Parallelism determines how many instances (tasks) of an operator run concurrently.

Key Points:

- Parallelism is set at the job, operator, or environment level.
- Total number of tasks = parallelism × number of operators.

Example:

- If parallelism = 3 for all operators in a pipeline with 3 operators (source, map, sink):
- Total tasks = 3 * 3 = 9 .

Operator Chaining - Operator chaining is an optimization in Flink where multiple operators are combined into a single task to reduce network overhead and improve efficiency. Operators are chained if they share the same parallelism.

```
stream = env.from_source(kafka_source).map(lambda x: x.upper()).filter(lambda x: "ERROR" in x)
stream.sink_to(kafka_sink)
```

Here, map and filter operators are chained into one task because:

- They share the same parallelism.
- They don't require data shuffling between them.

If parallelism = 4:

- There will be 4 tasks for the chained map and filter operators.
- If not chained, there would be 4 tasks for the map and 4 tasks for the filter, leading to 8 total tasks.

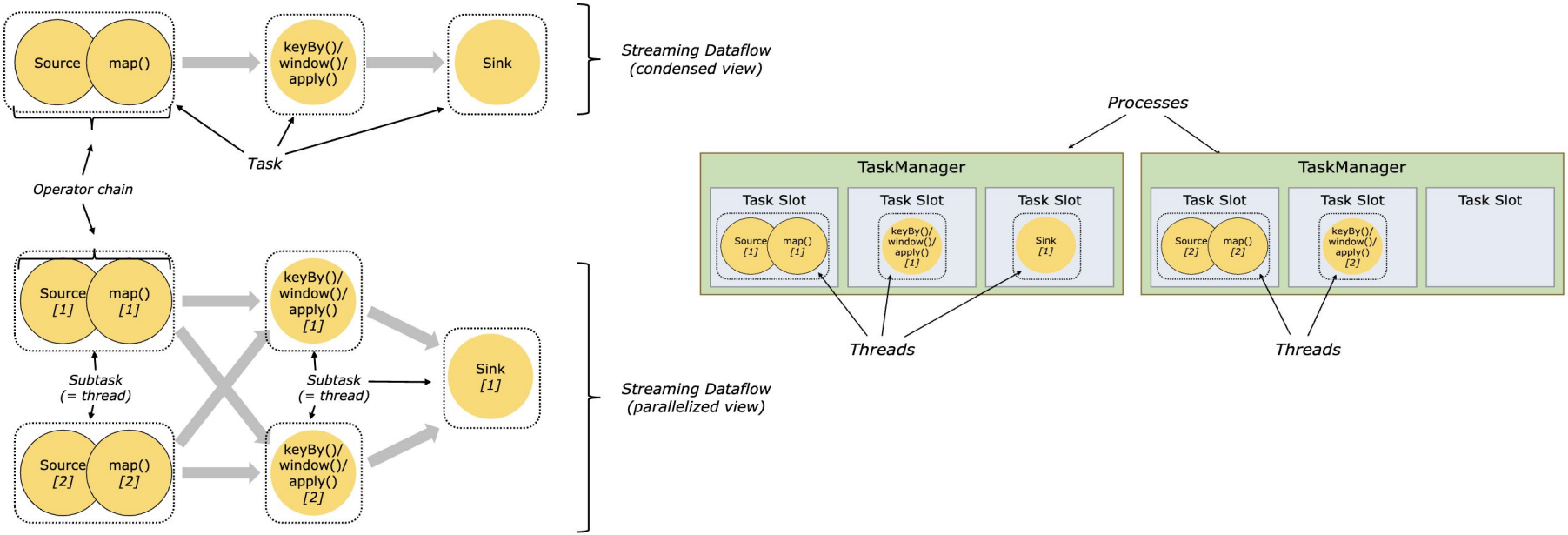
Apache Flink Architecture

Grow Data Skills

Task Slots - A task slot is the smallest unit of resource allocation (CPU, Memory, Network bandwidth) in a Task Manager. Each task slot can host one or more tasks.

Key Points:

- The total number of slots in a cluster = number of Task Managers × slots per Task Manager.
- Multiple tasks can run in a single slot if operator chaining is enabled.



Apache Flink Architecture

Grow Data Skills

Total Tasks on a Cluster

Cluster Example:

- Assume:
  - 2 Task Managers.
  - Each Task Manager has 4 slots.
  - Total slots in the cluster = 2 * 4 = 8 .
- Pipeline:
  - Source (parallelism = 2).
  - Map (parallelism = 4).
  - Sink (parallelism = 4).
- Tasks:
  - Source: 2 tasks.
  - Map: 4 tasks.
  - Sink: 4 tasks.
  - Total tasks = 2 + 4 + 4 = 10 .
- Slot Usage:
  - Each slot can handle multiple tasks due to operator chaining.

Apache Flink Architecture

Grow Data Skills

Task Manager and Slots - A Task Manager is a worker process that manages resources (CPU, memory) and executes tasks assigned to it. Each Task Manager is divided into slots.

Key Points:

- The number of tasks a Task Manager can execute depends on the number of slots it has.
- Slots are shared among multiple tasks only when chaining is enabled.

Example - If a Task Manager has 4 slots and the pipeline has 10 tasks:

- Tasks are distributed across slots in a round-robin fashion.
- Tasks may be chained to reduce the total number of active tasks.

State Management in Flink

Grow Data Skills

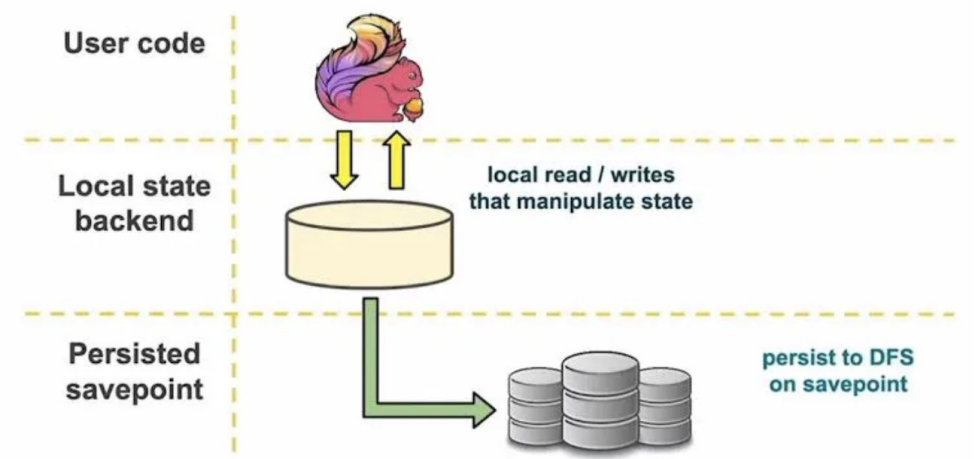
What is State?

In Apache Flink, state refers to data that is maintained across processing events in streaming or batch jobs. State allows Flink to keep track of information required to:

- Aggregate values (e.g., sum, average).
- Process events in a specific order.
- Enable fault tolerance by saving snapshots of state.

State can be classified into:

- ***Keyed State:*** Maintains state for specific keys (e.g., order ID, user ID).
- ***Operator State:*** Shared across operator instances (e.g., Kafka offsets in a source operator).



State Management in Flink

Grow Data Skills

State Backend - A state backend in Apache Flink is responsible for determining how and where the state of a Flink job is stored and managed. The choice of a state backend has significant implications for performance, scalability, and fault tolerance.

- ***HashMapStateBackend*** - The HashMapStateBackend stores state directly in the JVM heap memory of the TaskManager. It is a simple and lightweight backend best suited for small-scale use cases and testing.

  - Key Features:
    - In-Memory Storage: State is stored in the JVM's heap memory, making access and updates fast.
    - Checkpoint Storage: State snapshots are stored in an external distributed storage system, such as HDFS, S3, or any other compatible filesystem.
    - Use Case: Best suited for applications where the state is relatively small (e.g., key-value stores for counters or aggregates) and for local development and testing.

  - Advantages:
    - Faster read/write operations due to in-memory storage.
    - Simpler to set up and configure.

  - Limitations:
    - Limited scalability because it relies on the TaskManager's JVM heap size. Large states can cause OutOfMemoryErrors.
    - Full checkpoints are created during fault tolerance, which may become costly for larger states.

State Management in Flink

How to configure HashMapStateBackend?

=====> Configure in code

```
from pyflink.common.state_backend import HashMapStateBackend
env.set_state_backend(HashMapStateBackend())  # Configure HashMapStateBackend
```

=====> Configure in flink-conf.yaml

```
state.backend: hashmap
state.checkpoints.dir: s3://my-checkpoints/
state.savepoints.dir: s3://my-savepoints/
```

State Management in Flink

- ***RocksDBStateBackend*** - The RocksDBStateBackend stores state using **RocksDB**, a high-performance embedded key-value store that persists data to disk. Unlike the HashMapStateBackend, it is designed for handling large-scale applications and large states that exceed the JVM heap size.

  - Key Features:
    - Disk-Based Storage: State is stored on local disks, allowing the system to handle much larger states than the available JVM memory.
    - Incremental Checkpoints: Only changes (deltas) since the last checkpoint are stored, significantly reducing the size of checkpoints for large states.
    - Use Case: Ideal for production environments, high-volume streaming applications, and jobs requiring fault tolerance for large stateful computations.

  - Advantages:
    - Scalability: Handles state sizes much larger than the TaskManager's heap memory.
    - Better fault tolerance with incremental checkpointing, which minimizes checkpoint size and improves recovery time.

  - Limitations:
    - Disk I/O may make access and updates slower compared to in-memory backends.
    - Requires proper configuration of RocksDB to optimize performance.

State Management in Flink

How to configure RocksDBStateBackend?

=====> Configure in code

```
from pyflink.common.state_backend import EmbeddedRocksDBStateBackend
env.set_state_backend(EmbeddedRocksDBStateBackend())  # Configure RocksDBStateBackend
```

=====> Configure in flink-conf.yaml

```
state.backend: rocksdb
state.backend.rocksdb.localdir: /mnt/rocksdb
state.checkpoints.dir: s3://my-checkpoints/
state.savepoints.dir: s3://my-savepoints/
```

Checkpointing in Flink

Grow Data Skills

Checkpointing is a mechanism for fault tolerance in Apache Flink. It enables Flink to recover from failures and resume execution while ensuring data consistency. By periodically capturing the state of a streaming application, Flink can restart the job from the last checkpoint in case of failure, avoiding data loss or duplicate processing.

Why is Checkpointing Needed?

- ***Fault Tolerance*** - Checkpointing ensures that applications can recover and resume processing after a failure, such as TaskManager crashes or cluster restarts.
- ***Exactly-Once Processing*** - Flink ensures exactly-once state consistency by restoring state and replaying records from the checkpointed state, even during recovery.
- ***Durability*** - By persisting the state to external storage (e.g., HDFS, S3), Flink guarantees that the system can recover without relying on volatile memory.
- ***Recovery of Stateful Applications*** - Applications with state (e.g., aggregations, windows, keyed operators) require checkpointing to recover intermediate results and maintain correctness.

Checkpointing in Flink

Grow Data Skills

How Does Checkpointing Work?

- ***Triggering*** - A checkpoint is triggered periodically (based on the configured interval) by the JobManager.
- ***Operator Snapshot*** - Each operator in the job takes a snapshot of its state and sends it to a checkpoint coordinator.
- ***Persistence*** - The checkpoint data is written to a distributed storage system (e.g., HDFS, S3) for durability.
- ***Acknowledgment*** - After all tasks complete their snapshots, the checkpoint is acknowledged as successful.
- ***Recovery*** - Upon failure, Flink retrieves the latest checkpoint to restore the state and reprocess data from the saved offsets.

Types of Checkpointing

- ***Full Checkpointing:***
  - Captures the entire state of the application during every checkpoint.
  - Used in HashMapStateBackend.
  - Suitable for smaller states but can be expensive for large states.

- ***Incremental Checkpointing:***
  - Only the delta (changes) since the last checkpoint is stored.
  - Used in RocksDBStateBackend.
  - More efficient for large states because it minimizes the size of checkpoints.

Checkpointing in Flink

Grow Data Skills

Checkpointing mode defines the level of consistency Flink ensures during checkpointing. Flink offers two checkpointing modes:

- **EXACTLY_ONCE (Default)**
  - Guarantees that each record is processed exactly once, even in the case of failures.
  - Flink ensures state consistency by replaying records starting from the last successful checkpoint during recovery.
  - Ideal for scenarios requiring strict accuracy, such as financial or transaction processing systems.
  - How It Works:
    - Records that were in-flight during checkpointing are reprocessed after recovery to ensure all data is accounted for.
    - Works well with stateful transformations and connectors supporting exactly-once semantics (e.g., Kafka transactional writes).
  - Use Case:
    - Financial systems, fraud detection, and other critical applications requiring precision.

- **AT_LEAST_ONCE**
  - Guarantees that every record is processed at least once, but there might be duplicates during failure recovery.
  - Faster and less resource-intensive compared to EXACTLY_ONCE because it skips some additional guarantees like buffering in-flight records.
  - How It Works:
    - Records in the buffer or in transit might be replayed after recovery, leading to potential duplicates.
    - Suitable for use cases where occasional duplicates can be tolerated and eliminated downstream.
  - Use Case:
    - Log aggregation, monitoring, or analytics where duplicates are acceptable or post-processing is available to handle them.

Checkpointing in Flink

How to configure checkpointing in the code?

```
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.datastream.checkpoint_config import CheckpointingMode

# Create Stream Execution Environment
env = StreamExecutionEnvironment.get_execution_environment()

# Enable Checkpointing
env.enable_checkpointing(10000)  # Checkpoint every 10 seconds

# Set Checkpointing Mode
env.get_checkpoint_config().set_checkpointing_mode(CheckpointingMode.EXACTLY_ONCE)

# Configure Checkpoint Timeout
env.get_checkpoint_config().set_checkpoint_timeout(60000)  # 1 minute

# Configure Minimum Pause Between Checkpoints
env.get_checkpoint_config().set_min_pause_between_checkpoints(5000)  # 5 seconds

# Externalized Checkpoint Storage (e.g., S3 or HDFS)
env.get_checkpoint_config().set_checkpoint_storage("hdfs://namenode:8020/flink-checkpoints")
```

Checkpointing in Flink

How to configure checkpointing in the config?

```
state.backend: rocksdb                    # State backend to use
state.backend.incremental: true           # Enable incremental checkpointing (for RocksDB)
state.checkpoints.dir: s3://my-bucket/checkpoints/  # Storage location for checkpoints

execution.checkpointing.interval: 10s     # Trigger a checkpoint every 10 seconds
execution.checkpointing.timeout: 60s      # Maximum allowed time for checkpoint
execution.checkpointing.min-pause: 5s     # Minimum pause between checkpoints
execution.checkpointing.tolerable-failed-checkpoints: 3  # Tolerate 3 consecutive checkpoint failures
```

Savepointing in Flink

Grow Data Skills

A savepoint is a consistent snapshot of the state of a Flink application that you can use to stop, upgrade, or migrate your job. Unlike checkpoints, savepoints are manually triggered and designed for operational use cases, such as job upgrades or state migration between different Flink versions.

Savepoints differ from checkpoints in the following ways:

- ***Manually Triggered***: Savepoints are explicitly triggered by the user.
- ***Durable***: Savepoints are always stored in a persistent and user-defined location.
- ***Version Compatibility***: Savepoints allow you to upgrade your Flink job (e.g., deploy new versions of your application or update Flink's version).

Why is Savepointing Needed?

- ***Job Upgrades:*** Upgrade your Flink application while retaining the current state.
- ***Migration***: Move the application to a different cluster or Flink version.
- ***Graceful Stopping:*** Stop a job while ensuring no state is lost.
- ***Debugging***: Analyze the state of the job for debugging purposes.

Savepoint Lifecycle

- ***Trigger a Savepoint***: Manually create a savepoint to snapshot the state of the job.
- ***Stop the Job (if needed)***: Optionally stop the job after taking the savepoint.
- ***Resume from Savepoint***: Start the application from the savepoint when needed.

Savepointing in Flink

- **Stop and Savepoint** - To stop the job and take a savepoint in a single step:

  flink stop --savepointPath file:///flink/savepoints <jobId>

- **Resume Job from Savepoint** - Use the run command to restart the job from the savepoint:

  flink run -s file:///flink/savepoints/savepoint-12345 ./my-job.jar

Backpressure in Streaming Application

Grow Data Skills

What is Backpressure in Flink?

Backpressure occurs in Apache Flink when downstream operators cannot process data as quickly as upstream operators produce it. This creates a bottleneck in the data flow, as unprocessed data builds up in internal buffers, which can eventually lead to resource exhaustion.

How Backpressure Occurs ?

- ***High Data Volume*** - If a source operator ingests data at a rate higher than downstream operators can process, backpressure occurs.
- ***Inefficient Processing*** - A slow transformation or join operator may delay processing.
- ***Limited Resources*** - Insufficient CPU, memory, or I/O bandwidth for downstream tasks.
- ***Output Bottleneck*** - If a sink operator writes to a slow storage system or network, it may throttle the data flow.

How Backpressure Works in Flink ?

- ***Blocking Behavior*** - Flink's backpressure mechanism blocks upstream operators to prevent data overflow in buffers.
- ***Automatic Throttling*** - The upstream operator reduces its production rate to match the downstream operator's consumption rate.
- ***Metrics Monitoring*** - Flink provides metrics like "buffer pool usage" and "task latency" to monitor backpressure.

Backpressure in Streaming Application

Grow Data Skills

Checkpointing in Flink ensures fault tolerance by periodically saving the state of operators. However, backpressure significantly affects checkpointing:

- ***Delayed State Snapshot*** - Checkpoints capture the state of all tasks in the pipeline. If one task is experiencing backpressure, it slows down the entire checkpointing process.

- ***Increased Checkpoint Duration*** - Backpressure increases the time required to create consistent snapshots, as upstream tasks are blocked.

- ***Checkpoint Failures*** - If a task cannot acknowledge a checkpoint within the specified timeout due to backpressure, the checkpoint fails.

- ***Out-of-Memory Risks*** - Backpressure can lead to buffer overflows, which consume memory. If memory is exhausted during checkpointing, it can cause application failures.

How to Tackle Backpressure in Apache Flink ?

- ***Optimize Parallelism*** - Increase the parallelism of operators to distribute the workload more effectively across available task slots.
- ***Rate Limiting at Source*** - Throttle the data ingestion rate at the source to prevent overwhelming downstream operators.
- ***Increase Buffer Size*** - Adjust the network buffer size to allow operators to handle temporary spikes in data volume.
  Configure buffer memory in flink-conf.yaml:

  taskmanager.network.memory.fraction: 0.2 # Allocate 20% of memory to network buffers
  taskmanager.network.memory.min: 128mb
  taskmanager.network.memory.max: 2gb

Backpressure in Streaming Application

- ***Monitor and Adjust Resources Dynamically*** - Use Flink's metrics to identify bottlenecks and dynamically adjust resources (e.g., CPU, memory, task slots). Monitor the "Backpressure Ratio" in the Flink Web UI and scale resources accordingly.

- ***Optimize Checkpointing*** - Minimize the impact of checkpointing on performance by configuring it effectively. Increase checkpoint intervals and timeouts: