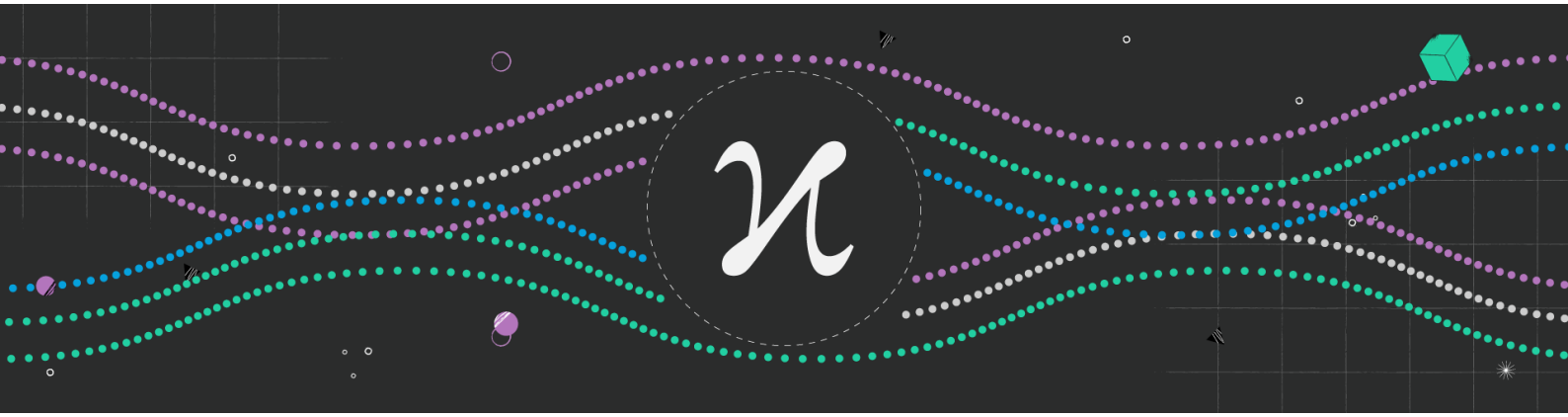
Big Data Architecture

Kappa Architecture - Building a Modern Streaming Data Architecture



Made By :

- **M. BOUADIF A**bdelkrim
- **M. FAHMI O**thmane
- **M. IDRISSI H**amza

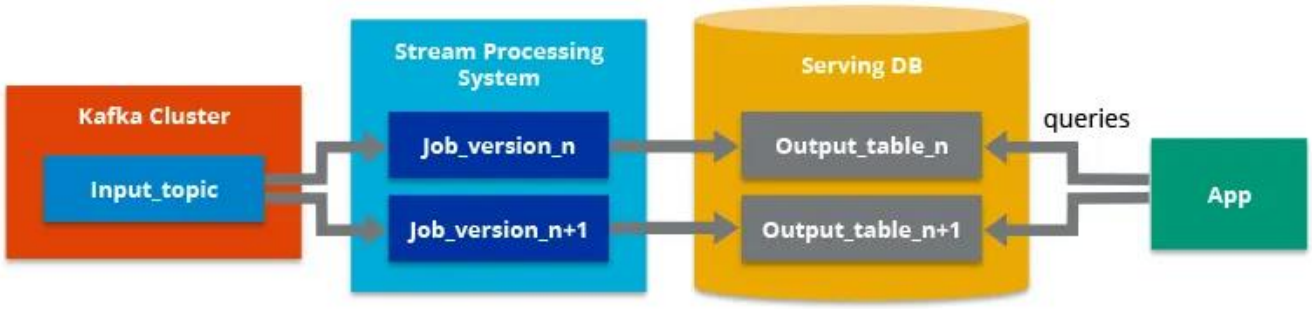
Supervised By :

- **Pr. EL YUSUFI Y**asyn

2023/2025

Introducing the Kappa Architecture

Kappa Architecture represents a shift in the way we approach data processing architectures. Developed as a response to the challenges posed by Lambda architecture, Kappa proposes a simpler, more streamlined approach. The primary aim of Kappa Architecture is to process streaming data in a way that provides timely insights, reduces system complexity, and ensures data consistency. It achieves this by focusing on one core principle: treating all data as a stream.



Tools

STREAMING Layer

An append-only immutable log store is the canonical store in a Kappa Architecture (or Lambda Architecture) system. Some log databases:

- Amazon Quantum Ledger Database (QLDB)
- **Apache Kafka**
- Apache Pulsar
- Amazon Kinesis
- Amazon DynamoDB Streams
- Azure Cosmos DB Change Feed
- Azure EventHub
- DistributedLog
- EventStore
- Chronicle Queue
- Pravega

Serving Layer

In Kappa Architecture, data is fed from the log store into a streaming computation system. Some distributed streaming systems:

- Amazon Kinesis
- **Apache Flink**
- Apache Samza
- Apache Spark
- Apache Storm
- Apache Beam
- Azure Stream Analytics
- Hazelcast Jet
- Kafka Streams
- Onyx
- Siddhi
- Materialize

Getting Started

In this document we go over how to use Apache Flink Table API in Python to consume data from and write data to a Confluent Community Platform Apache Kafka Cluster running locally in Docker. Apache Flink is a highly scalable and performant computing framework for performing stateful streaming computation with exactly once processing semantics. Apache Kafka is a durable, low latency, replayable log based message queue popular in distributed architectures. Together these technologies facilitate developing advanced real-time analytics and loosely coupled event driven systems.

Plan

Apache Flink Table API Sources and Sinks

Apache Flink's Table API uses constructs referred to as table sources and table sinks to connect to external storage systems such as files, databases, and message queues. Table sources are conduits through which Apache Flink consumes data from external systems. Similarly, table sinks are the conduit through which data is emitted from Apache Flink to external systems.

Setting Up Confluent Kafka in Local Docker Environment

For quickly launching a small development instance of Kafka we often piggyback on the work of the fine folks over at Confluent who graciously distribute Community and Enterprise Platform Docker Images of Kafka and Kafka related infrastructure.

Below is a docker-compose.yaml file which we use to spin up a local Kafka cluster for this demonstration.

```
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:6.1.1
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-kafka:6.1.1
    hostname: broker
    container_name: broker
    depends_on:
      - zookeeper
    ports:
      - "29092:29092"
      - "9092:9092"
      - "9101:9101"
```

```
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
```

To run this Docker Compose service we run the following in the same directory as the above shown docker-compose.yaml file.

>>>docker-compose up -d

```
PS C:\Users\DR2\Desktop\IASD\S2\BigData\Architecture-Kappa> docker-compose
up -d
[+] Running 2/3
 - Network architecture-kappa_default  Created                    8.9s
 ✓ Container zookeeper                 Start...                   6.3s
 ✓ Container broker                    Started                    5.8s
PS C:\Users\DR2\Desktop\IASD\S2\BigData\Architecture-Kappa>
```

To establish source and sink tables in the Apache Flink program WE ll create two topics in Kafka. The first topic will be named sales-usd which will hold fictitious sales data with sales values in US Dollars and be used as input data to my Flink program. The second topic will be named sales-euros and be where we write the sales in USD converted to Euros.

To create source Kafka topic we use the kafka-topics CLI available inside the Confluent Kafka container named broker.

```
>>>docker exec -it broker kafka-topics --create --bootstrap-server
localhost:9092 --topic sales-usd
```

Then we similarly create the destination Kafka topic.

```
>>>docker exec -it broker kafka-topics --create --bootstrap-server
localhost:9092 --topic sales-euros
```

The last piece of the Kafka setup to go over is the Python based sales producer program named sales_producer.py which WE ll use to generate fake sales data to work with in this tutorial. The sales_producer.py source is listed below.

```
import argparse
import atexit
import json
import logging
import random
import time
import sys


from confluent_kafka import Producer


logging.basicConfig(
```

```
  format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
  datefmt='%Y-%m-%d %H:%M:%S',
  level=logging.INFO,
  handlers=[
      logging.FileHandler("sales_producer.log"),
      logging.StreamHandler(sys.stdout)
  ]
)

logger = logging.getLogger()

SELLERS = ['LNK', 'OMA', 'KC', 'DEN']


class ProducerCallback:
    def __init__(self, record, log_success=False):
        self.record = record
        self.log_success = log_success

    def __call__(self, err, msg):
        if err:
            logger.error('Error producing record {}'.format(self.record))
        elif self.log_success:
            logger.info('Produced {} to topic {} partition {} offset {}'.format(
                self.record,
                msg.topic(),
                msg.partition(),
                msg.offset()
            ))


def main(args):
    logger.info('Starting sales producer')
    conf = {
        'bootstrap.servers': args.bootstrap_server,
        'linger.ms': 200,
        'client.id': 'sales-1',
        'partitioner': 'murmur2_random'
    }

    producer = Producer(conf)


    atexit.register(lambda p: p.flush(), producer)

    i = 1
    while True:
        is_tenth = i % 10 == 0

        sales = {
            'seller_id': random.choice(SELLERS),
            'amount_usd': random.randrange(100, 1000),
            'sale_ts': int(time.time() * 1000)
        }
```

```
        producer.produce(topic=args.topic,
                         value=json.dumps(sales),
                         on_delivery=ProducerCallback(sales, log_success=is_tenth))

        if is_tenth:
            producer.poll(1)
            time.sleep(5)
            i = 0 # no need to let i grow unnecessarily large

        i += 1


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--bootstrap-server', default='localhost:9092')
    parser.add_argument('--topic', default='sales-usd')
    args = parser.parse_args()
    main(args)
```

Flink Python Sales Processor Application

When it comes to connecting to Kafka source and sink topics via the Table API we have two options. we can use the Kafka descriptor class to specify the connection properties, format and schema of the data or we can use SQL Data Definition Language (DDL) to do the same. we prefer the later as we find the DDL to be more readable and a slightly more stable API.

Before we can read or write data from an external Kafka cluster we need to download the Kafka SQL Connector so we can configure my Apache Flink program to use it. The Kafka SQL Connector is a simple Jar library which we can download with a HTTP Client such as HTTPie.

>>>python -m pip install httpie

>>>http --download https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-kafka_2.11/1.13.0/flink-sql-connector-kafka_2.11-1.13.0.jar

NB: Python version (3.6, 3.7 or 3.8) is required for PyFlink.

Move the .jar file to the Lib of your python kernel!!!!!

>>>C:\Users\DR2\AppData\Local\Programs\Python\Python38\Lib\site-packages\pyflink\lib\flink-sql-connector-kafka_2.11-1.13.0.jar

Now in the same directory as the Jar file we just download WE will create a new python module named sales_processor.py representing the Flink program. The contents of the PyFlink program are shown below.

```
import os

from pyflink.datastream import StreamExecutionEnvironment
```

```
from pyflink.table import StreamTableEnvironment, EnvironmentSettings


def main():
    # Create streaming environment
    env = StreamExecutionEnvironment.get_execution_environment()

    settings = EnvironmentSettings.new_instance()\
                      .in_streaming_mode()\
                      .use_blink_planner()\
                      .build()

    # create table environment
    tbl_env = StreamTableEnvironment.create(stream_execution_environment=env,
                                            environment_settings=settings)

    # add kafka connector dependency
    kafka_jar = os.path.join(os.path.abspath(os.path.dirname(__file__)),
                             'flink-sql-connector-kafka_2.11-1.13.0.jar')

    tbl_env.get_config()\
            .get_configuration()\
            .set_string("pipeline.jars", "file:///{}".format(kafka_jar))

    ###########################################################################
    # Create Kafka Source Table with DDL
    ###########################################################################
    src_ddl = """
        CREATE TABLE sales_usd (
            seller_id VARCHAR,
            amount_usd DOUBLE,
            sale_ts BIGINT,
            proctime AS PROCTIME()
        ) WITH (
            'connector' = 'kafka',
            'topic' = 'sales-usd',
            'properties.bootstrap.servers' = 'localhost:9092',
            'properties.group.id' = 'sales-usd',
            'format' = 'json'
        )
    """
    import sys
    print(sys.path)
    tbl_env.execute_sql(src_ddl)

    # create and initiate loading of source Table
    tbl = tbl_env.from_path('sales_usd')

    print('\nSource Schema')
    tbl.print_schema()

    ###########################################################################
    # Define Tumbling Window Aggregate Calculation (Seller Sales Per Minute)
    ###########################################################################
```

```
    sql = """
        SELECT
          seller_id,
          TUMBLE_END(proctime, INTERVAL '6' SECONDS) AS window_end,
          SUM(amount_usd) * 0.85 AS window_sales
        FROM sales_usd
        GROUP BY
          TUMBLE(proctime, INTERVAL '6' SECONDS),
          seller_id
    """
    revenue_tbl = tbl_env.sql_query(sql)

    print('\nProcess Sink Schema')
    revenue_tbl.print_schema()

    ################################################################
    # Create Kafka Sink Table
    ################################################################
    sink_ddl = """
        CREATE TABLE sales_euros (
            seller_id VARCHAR,
            window_end TIMESTAMP(3),
            window_sales DOUBLE
        ) WITH (
            'connector' = 'kafka',
            'topic' = 'sales-euros',
            'properties.bootstrap.servers' = 'localhost:9092',
            'format' = 'json'
        )
    """
    tbl_env.execute_sql(sink_ddl)

    # write time windowed aggregations to sink table
    revenue_tbl.execute_insert('sales_euros').wait()

    tbl_env.execute('windowed-sales-euros')


if __name__ == '__main__':
    main()
```

The source code should be reasonably commented enough to get a pretty good idea of what is going on. Briefly, we establish a TableEnvironment in streaming mode which is the main entry point through which my Flink processing code is funneled into the Flink runtime. After creating the TableEnvironment we register the Kafka SQL Connector Jar library with the environment. Next we use DDL to define the source table to map the schema and subsequent data of the Kafka input topic to the Schema of an associated Flink table.

With the TableEnvironment configured to source data from Kafka we then write a simple SQL query to perform a one minute tumbling windowed aggregate to sum and convert seller sales from USD to Euros. The results of this calculation is assigned to the Table variable named revenue_tbl. The last thing the program does is define a sink Flink table mapping the results of the revenue aggregation and conversion calculation to a destination Kafka topic named sales-euros.

Running the App

In this section we show the steps required to get this demo application running. First things first we always recommend creating a Python virtual environment which we will use the venv module that ships as part of my Python 3.8 version WE m working with.

```
>>>venv\Scripts\activate.bat
```

Then use pip to install confluent-kafka Python client library to write demo data into sales-usd Kafka topic. we also install the apache-flink Python Flink library to code my PyFlink program against.

```
>>>pip install "apache-flink>=1.13.0,<1.14" "confluent-kafka>=1.7.0,<1.8"
```

To run this app WE ll need to use two active terminals at once, both of which must have the Python virtual environment we setup and installed the confluent-kafka and apache-flink libraries in previously.

In the first terminal we launch the producer.py program to begin generating fake sales data to work with in the processor.py Flink program.

```
>>>python producer.py
```

```
PS C:\Users\DR2\Desktop\IASD\S2\BigData\Architecture-Kappa> python sales_pr
oducer.py
2024-04-21 00:14:44 root         INFO     Starting sales producer
2024-04-21 00:14:51 root         INFO     Produced {'seller_id': 'OMA', 'am
ount_usd': 155, 'sale_ts': 1713654885121} to topic sales-usd partition 0 of
fset 9
2024-04-21 00:14:56 root         INFO     Produced {'seller_id': 'KC', 'amo
unt_usd': 805, 'sale_ts': 1713654891148} to topic sales-usd partition 0 off
set 19
```

Then in second terminal we launch the processor.py Flink program which consumes the fake sales data from the sales-usd Kafka topic, performs the revenue aggregation and conversion to Euros and writes the results to the destination Kafka topic sales-euros.

```
>>>python processor.py
```

```
PS C:\Users\DR2\Desktop\IASD\S2\BigData\Architecture-Kappa> python sales_pr
ocessor.py
['C:\\Users\\DR2\\Desktop\\IASD\\S2\\BigData\\Architecture-Kappa', 'C:\\Use
rs\\DR2\\AppData\\Local\\Programs\\Python\\Python38\\python38.zip', 'C:\\Us
ers\\DR2\\AppData\\Local\\Programs\\Python\\Python38\\DLLs', 'C:\\Users\\DR
2\\AppData\\Local\\Programs\\Python\\Python38\\lib', 'C:\\Users\\DR2\\AppDa
ta\\Local\\Programs\\Python\\Python38', 'C:\\Users\\DR2\\AppData\\Local\\Pr
ograms\\Python\\Python38\\lib\\site-packages']

Source Schema
(
  `seller_id` STRING,
  `amount_usd` DOUBLE,
  `sale_ts` BIGINT,
  `proctime` TIMESTAMP_LTZ(3) NOT NULL *PROCTIME* AS PROCTIME()
)

Process Sink Schema
(
  `seller_id` STRING,
  `window_end` TIMESTAMP(3) NOT NULL,
  `window_sales` DOUBLE
)
```

After waiting a couple of minutes to allow a few one minutes tumbling windows to elapse we can fire up a kafka-console-consumer instance which is available within the Kafka broker container to prove that data is flowing from source to sink Kafka topics.

```
>>>docker exec -it broker kafka-console-consumer --from-beginning --bootstrap-server localhost:9092 --topic sales-euros
```

```
PS C:\Users\DR2\Desktop\IASD\S2\BigData\Architecture-Kappa> docker exec -i
 broker kafka-console-consumer --from-beginning --bootstrap-server localho
t:9092 --topic sales-euros
{"seller_id":"LNK","window_end":"2024-04-21 00:17:42","window_sales":1717.
}
{"seller_id":"DEN","window_end":"2024-04-21 00:17:42","window_sales":3182.
}
{"seller_id":"KC","window_end":"2024-04-21 00:17:42","window_sales":3646.5
{"seller_id":"OMA","window_end":"2024-04-21 00:17:42","window_sales":2067.
}
{"seller_id":"LNK","window_end":"2024-04-21 00:17:48","window_sales":1090.
5}
```

References

1. https://milinda.pathirage.org/kappa-architecture.com/
2. https://thecodinginterface.com/blog/kafka-source-sink-with-apache-flink-table-api/
3. https://nexocode.com/blog/posts/kappa-architecture/
4. https://kavitmht.medium.com/building-a-real-time-data-streaming-pipeline-using-apache-kafka-flink-and-postgres-a22101c97895
5. https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/dev/python/installation/