DATA STREAMING: KAFKA DOCUMENTATION

1. WHY Kafka?

Imagine a scenario where a delivery company needs to track the location of its drivers every 10 seconds. This requires frequent calls to *getCurrentLocation()* and sending the data to a server. The database stores this information for future analytics, and another server accesses the database to forward the data to customers.

**The Problem**:

* High-throughput: With 1000 drivers in a city, the system generates 12,000 operations per minute (12 OPS/minute/driver \* 1000 drivers).
* Real-time processing: The system requires processing and forwarding data to customers in real-time.

**The Challenge**:

* Traditional databases cannot handle such high-throughput and real-time processing demands, leading to:
  + Slow performance
  + Increased latency
  + System crashes



How Kafka Helps:

* High-throughput handling: Kafka can process **over 2 million messages per second**, far exceeding the **25,000 messages per second that a traditional database** can handle.
* Real-time data streaming: Kafka is designed for **real-time data streaming**, allowing for **immediate processing and forwarding of data**.
* Complementary to databases: Kafka is **not a replacement for databases**, but rather a complementary tool. Databases provide persistent storage, while Kafka handles high-throughput and real-time data streaming.
* Decoupling producers and consumers: Kafka enables producers (drivers) to send data without waiting for consumers (servers) to process it, reducing the load on the database.



**KAFKA ARCHITECTURE:**

Key Components:

* **Producers**: Sends data to Kafka topics.
* **Topics**: Logical streams of data with dedicated brokers.
* **Broker**: A single instance of kafka server, managing and storing topic partitions.
* **Partition**: Ordered, immutable logs of data within a topic.
* **Consumer**: Subscribes to a topic and reads data from the partition.
* **Consumer Group**: Groups of customers that read the same kind of data.
* **Replication factor**: The number of copies of a partition that kafka maintains. Leader broker receives data from producers and follower brokers replicate the data. Ensuring even if the leader goes down; streaming service is maintained.





**QUIXSTREAMS [ Python Library]**

**What options exist:**

Python Kafka Client Libraries:

* Lacks advanced data engineering capabilities
* Missing key processing primitives (filtering, joining)
* Essential for stream processing (e.g., Kafka Streams)
* Limits sophisticated real-time data manipulation, transformation

Kafka REST APIs (Confluent REST Proxy):

* Facilitates easy Kafka interaction (HTTP)
* Suitable for gluing interfaces together
* Suboptimal for ML workloads (low latency, strict SLAs)
* Overhead from RESTful communication
* Stream processing with native Kafka clients offers better performance for real-time ML applications

**What is QuixStreams?**

* Quix Streams is a stream processing library focused on ease of use for Python data engineers.
* Instead of a database, Quix Streams uses a data streaming platform such as Apache Kafka. You can process data with high performance and save resources without introducing a delay.
* Lightweight, powerful, no JVM and no need for separate clusters of orchestrators.
* Does not use any domain-specific language or embedded framework

**The QuixStreams API:**

**The Good (Quix API and Tooling)**

* Data engineer-focused API/tooling
* Not directly competing with Java developers (e.g., Kafka Streams)
* Microservices/data mesh approach (application/data product per use case)
* Choose the right tool for the job

API Characteristics:

* Mostly sleek
* Some unintuitive parts (still in beta)
* Expected to improve in future releases

Integration:

* Excellent with Python data engineering/ML frameworks (e.g., Pandas, NumPy)
* Stream processing combined with Python libraries enhances developer experience

Compatibility:

* Works with open-source Kafka.

**The Improvable (Quix API)**

* Stream processing capabilities limited (e.g., sliding windows)
* Not comparable to advanced engines (Kafka Streams, Apache Flink)
* Enhanced features planned in the roadmap

Architecture:

* Complex architecture (Python -> C# -> C++)
* Relevance depends on use case, security, etc.

Performance Considerations:

* Benchmark comparison with Faust (Python-native) could be insightful
* Trade-off: inter-language marshaling/unmarshalling vs. performance of lower-level compiled languages

#### **Should one use Quix Streams or Apache Flink?**

* Apache Flink:
  + Java-first approach
  + Python support added later (DataStream, Table APIs)
  + Mature, with customer case studies
  + Use Flink if the use case makes sense with SQL or Java. And if the team is willing to operate its own Flink cluster or has a platform team or a cloud service taking over the operational burden and complexity.
* QuixStream:
  + Brand new, lacks maturity and case studies
  + Great choice for some Python-based stream processing projects
  + Use Quix Streams for Python projects if you want to go to production with a more microservice-like architecture building Python applications.