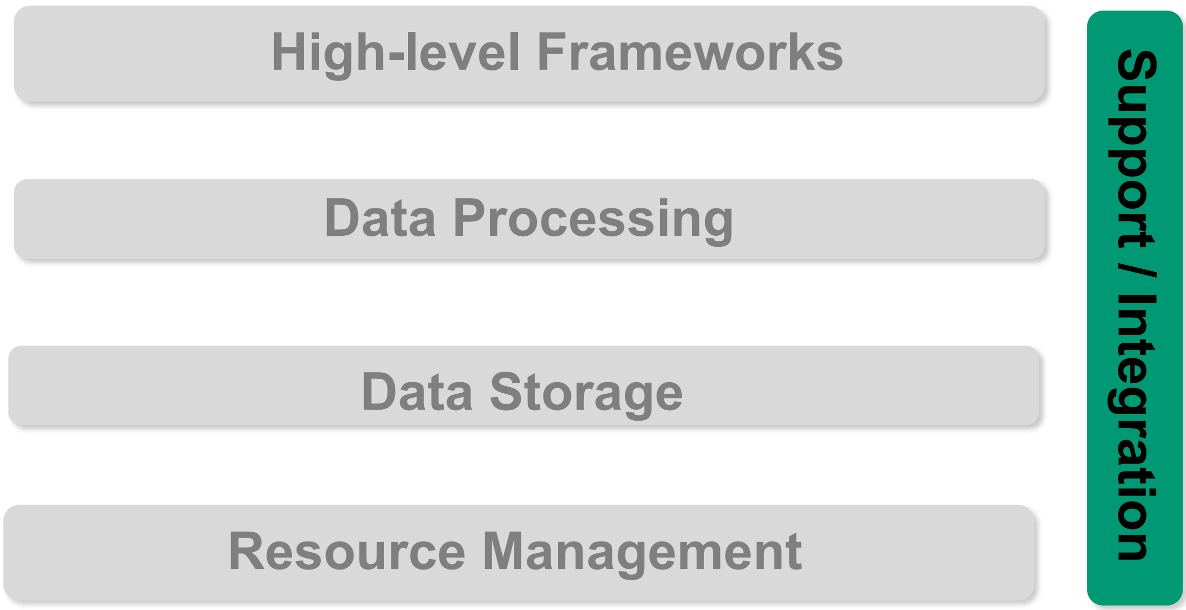
TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Data Acquisition and Ingestion

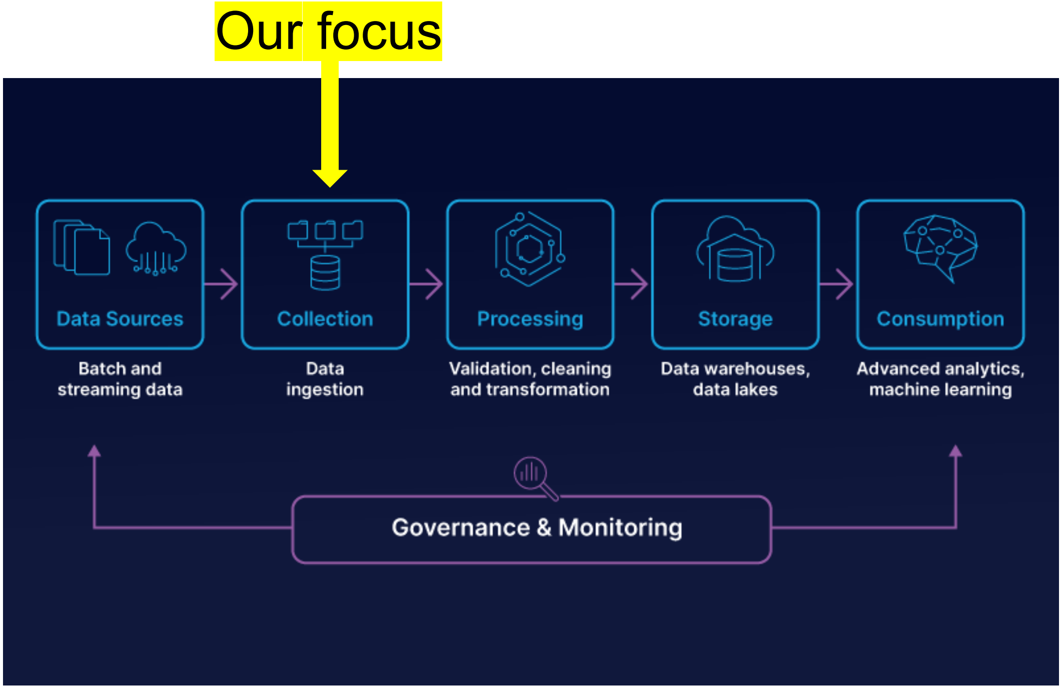
Corso di Sistemi e Architetture per Big Data
A.A. 2024/25
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Components of a data pipeline



Source: https://www.striim.com/blog/guide-to-data-pipelines/

Data ingestion

- How to collect data from external and multiple data sources and ingest it into target system where it can be stored and later analyzed?
  - For now: distributed file systems, NoSQL data stores, batch processing frameworks
- How to connect external data sources to stream or in-memory processing systems for immediate use?
- How and where to perform data preprocessing (e.g., data transformation, data conversion)?
- Data ingestion pipeline goal: move data - either batched or streaming - from multiple sources to a target destination, making it available for further processing and analysis

Driving factors

- Data source type and location
  - Data source: where data originates
  - Batch data sources: files, logs, RDBMS, …
  - Real-time data sources: IoT sensors, social media feeds, stock market feeds, …
  - Source location
- Velocity
  - How fast data is generated?
  - How frequently data varies?
  - Real-time or streaming data require low latency and low overhead
- Ingestion mechanism
  - Depends on data consumer
  - Pull vs. push based approach

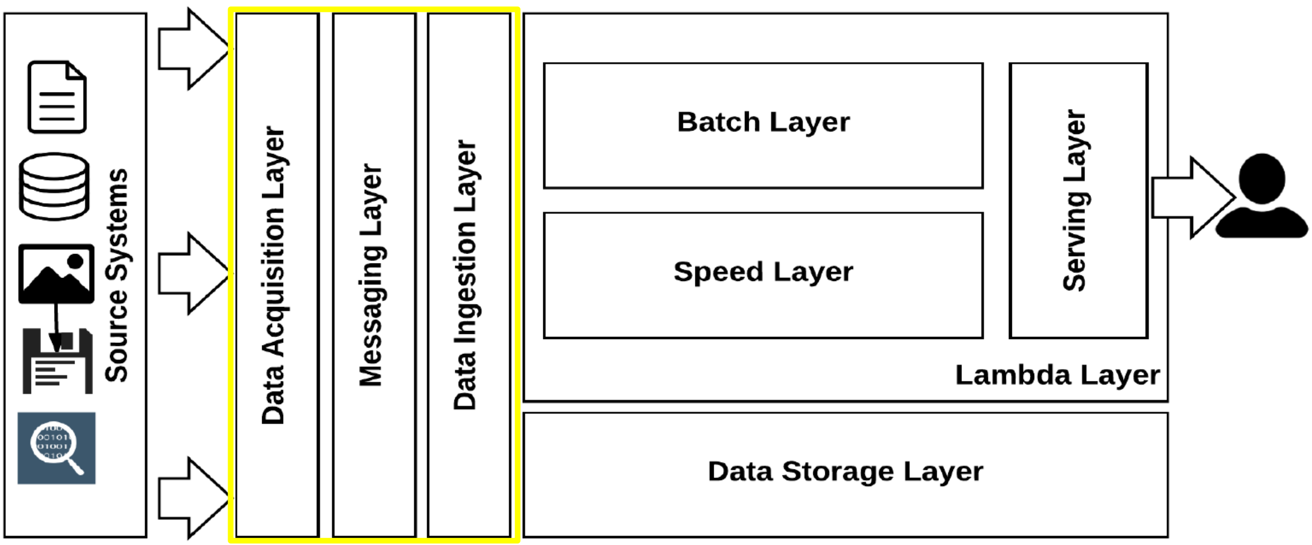
Requirements for data acquisition and ingestion

- Ingestion
  - Batch data, streaming data
  - Easy writing to storage (e.g., HDFS)
- Decoupling
  - Separate data sources from processing
- High availability and fault tolerance
  - Data ingestion available 24x7
  - For streaming data: buffering (persistence) in case processing framework is not available
- Scalability and high throughput
  - Number of sources and consumers will increase, amount of data will increase
- Data provenance
  - Track where data came from, how it was transformed, and how it flows through various systems

Requirements for data acquisition and ingestion

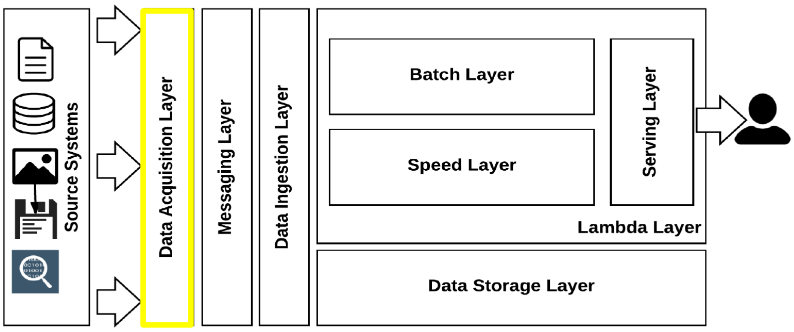
- Security
  - Data authentication and encryption
- Data conversion
  - From multiple sources: transform data into common format
  - Also to speed up processing
- Data integration
  - From multiple flows to single flow
- Data preprocessing
  - Raw data is transformed, cleaned, and prepared for analysis (e.g., range checks, filtering, missing data handling)
- Data compression
- Data routing
- Backpressure
  - Data buffering in case of temporary spikes in workload, so that data can be replayed later without loss

A unifying view



Data acquisition layer

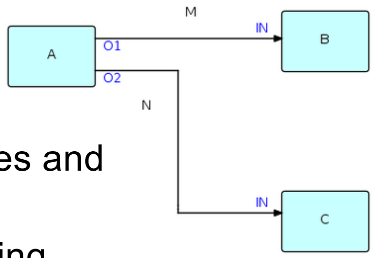
- Allows collecting, aggregating and moving data
- From various sources (server logs, social media, IoT sensors, …)
- To a data store (messaging system, distributed file system, NoSQL data store)
- We analyze
  - **Apache NiFi**



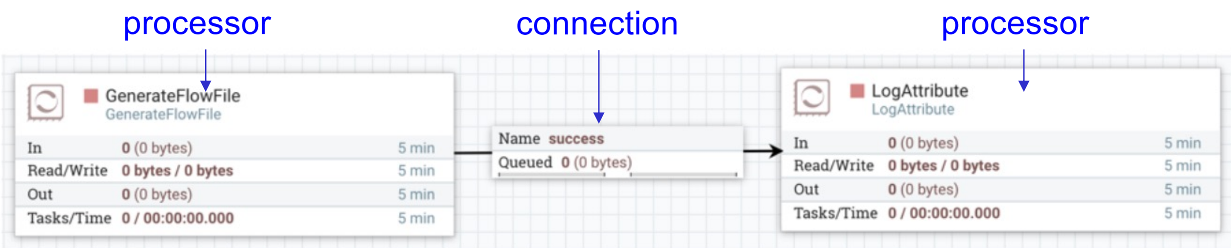
Apache NiFi

- Easy to use, powerful and reliable system to automate the flow of data between systems, mainly used for data routing and transformation https://nifi.apache.org
- Highly configurable
  - Flow specific QoS: loss-tolerant vs guaranteed delivery, low latency vs high throughput
  - Dynamic prioritization of queues
  - Flow can be modified at runtime: useful for preprocessing
  - Backpressure control
- Ease of use: drag-and-drop web-based UI to create, manage and monitor the dataflow
  - Allows to define sources from where to collect data, processors for data transformation, destinations to store data
- Data provenance and security (SSL, data encryption)

NiFi: core concepts



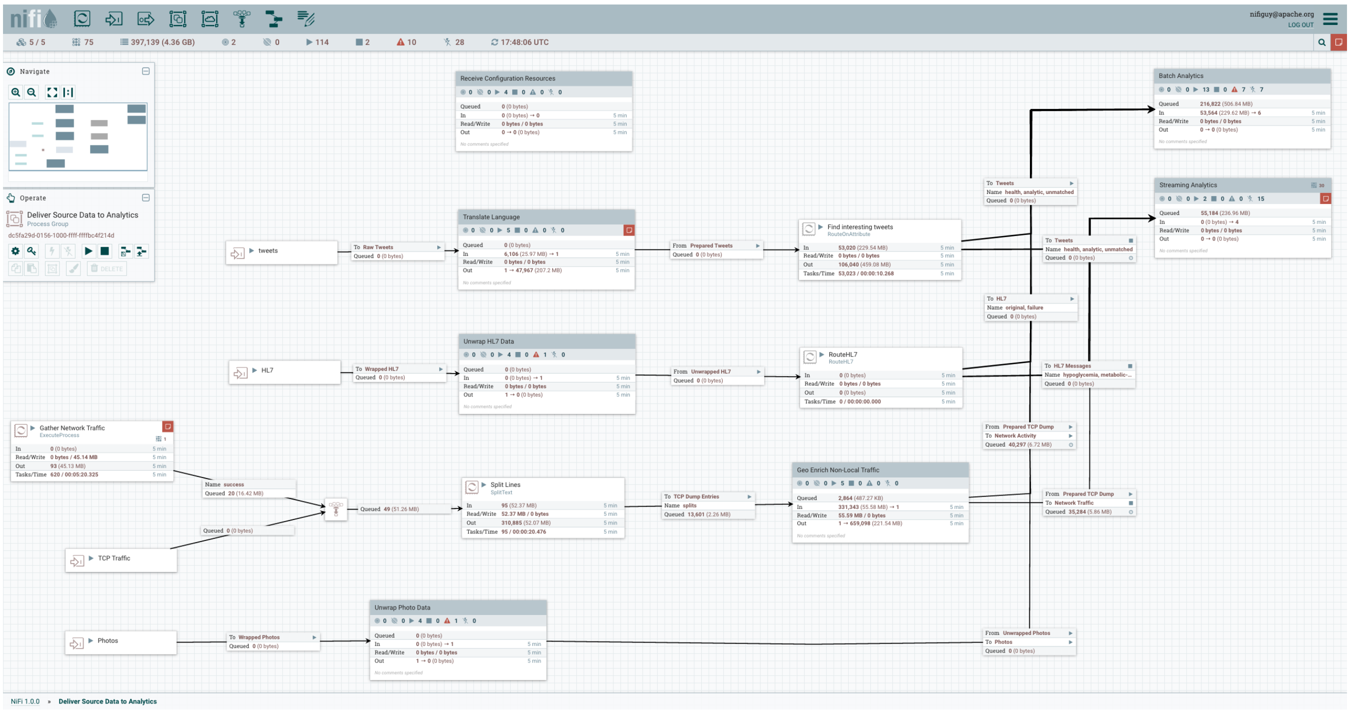
- Based on flow-based programming
- Main NiFi concepts:
  - FlowFile: piece of user data, made of attributes and content
  - FlowFile Processor: performs the work (sending, receiving, transforming, routing, splitting, merging, and processing FlowFiles)
  - Connection: defines how data flows from one Processor to another; has a queue FlowFiles are stored temporarily until the next Processor or destination can process them



NiFi: visual command & control

- Drag and drop Processors to build a flow
  https://nifi.apache.org/docs/nifi-docs/html/getting-started.html
- Start, stop and configure components in real time
- View errors and corresponding messages
- View statistics and health of data flow
- Create templates (i.e., reusable sub-flows) for common Processors and Connections

NiFi: visual command & control



NiFi: processors

- Main steps to create and run the dataflow
  - Add Processors
  - Configure Processors
  - Connect Processors among them
  - Start and stop Processors
  - Get info on Processors

NiFi: processors

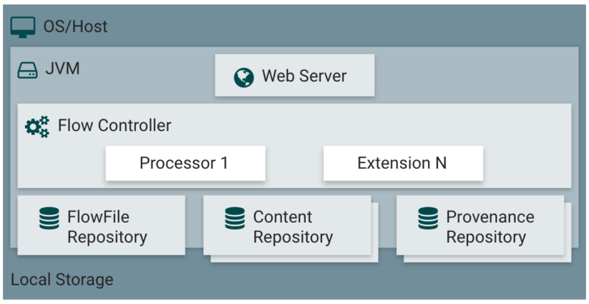
- NiFi provides many different Processors out of the box
  - Capabilities to ingest data from many different systems, route, transform, process, split, and aggregate data, and distribute data to many systems
  - Classified by category
- Data transformation
  - E.g., `CompressContent`, `EncryptContent`, `ReplaceText`
- Routing and mediation
  - E.g., `ControlRate`, `DistributeLoad`, `RouteOnContent`
- Database access
  - E.g., `ExecuteSQL`, `PutSQL`
- Attribute extraction
  - E.g., `ExtractText`, `HashContent`, `IdentifyMimeType`

NiFi: processors

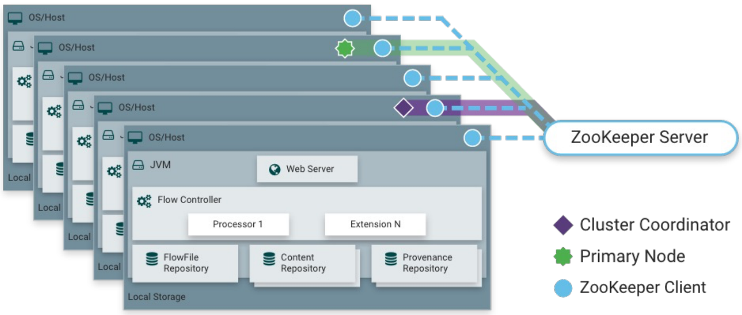
- System interaction
  - E.g., `ExecuteProcess`
- Data ingestion
  - E.g., `GetFile`, `GetFTP`, `GetHTTP`, `ListenUDP`, `GetHDFS`, `FetchS3Object`, `ConsumeKafka`, `GetMongo`, `GetTwitter`
- Data egress / Sending data
  - E.g., `PutEmail`, `PutFile`, `PutFTP`, `PutHDFS`, `PutSQL`, `PublishKafka`, `PutMongo`
- Splitting and aggregation
  - E.g., `SplitText`, `UnpackContent`, `MergeContent`, `SplitContent`
- HTTP
  - E.g., `GetHTTP`, `PostHTTP`, `InvokeHTTP`, `ListenHTTP`
- Amazon Web Services
  - E.g., `FetchS3Object`, `PutS3Object`, `GetSQS`, `PutSQS`

NiFi: architecture

- NiFi executes within a JVM



- Multiple NiFi servers can be clustered for scalability



NiFi: use case

- Use NiFi to fetch tweets by means of NiFi's processor 'GetTwitter'
  - Use Twitter Streaming API to retrieve tweets
- Move data stream to Apache Kafka using NiFi's processor 'PublishKafka'

Data serialization formats for Big Data

- Serialization: process of converting structured data into a compact (binary) form
- Data serialization formats you already know
  - JSON
  - Protocol buffers
- Other serialization formats
  - Apache Avro (row-oriented)
  - Apache ORC (column oriented)
  - Apache Parquet (column-oriented)
  - Apache Thrift https://thrift.apache.org

Choice of data serialization format

- Impacts various aspects of data processing, including efficiency, performance, and compatibility
- Efficiency: smaller size (data storage and transfer)
- Performance: faster reads, faster writes
- Compatibility: data can be shared across different systems and applications
- Support for schema evolution
  - Changes in data structure over time without breaking compatibility with older versions
- Support for splitting large files
  - Break up large files into smaller pieces that are easier to store, transfer, and process
- Advanced compression

Apache Avro

- Key features https://avro.apache.org/
  – Compact, binary, row-based data format
  – Relies on schema: data+schema is fully self-describing
    - Schema is defined in JSON and segregated from data
    - Supports schema evolution
  – Supports rich data types, including complex structures and nested objects
  – Supports compression techniques
    - Including Snappy, Deflate, and Bzip2
  – Cross and multi-language
    - Data can be serialized in one language and deserialized in another
    - Simple integration with dynamic languages
    - Can be used in RPC
  – Spark (and Hadoop) can access Avro as data source
  https://spark.apache.org/docs/latest/sql-data-sources-avro.html

Apache Avro

- Performance
  – Impact of serialization and deserialization times on small objects, ProtoBuf is faster
  – Storage efficiency for large data files lower than Parquet
    - Avro uses compact binary encoding, but no columnar compression techniques as Parquet
  – Optimized for writes, not for reads
    - Avro performs better than Parquet for write-intensive workloads
    - Scanning large datasets requires reading entire records, leading to slower query performance in analytics workloads
    - Parquet is preferable for analytical queries
  – Better than Parquet when schema evolution is frequent

https://www.datacamp.com/blog/avro-vs-parquet

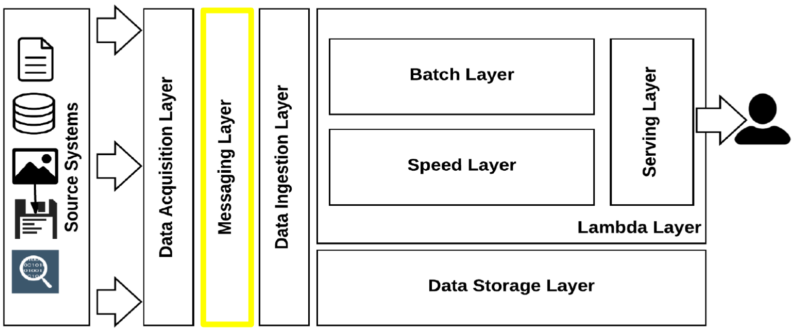
Apache ORC

- ORC (Optimized Row Columnar): format optimized for analytics workloads https://orc.apache.org/
- Key features
  - Columnar storage
  - Compression efficiency
    - Multiple codecs, including Snappy, zlib
    - Lightweight compression techniques such as dictionary encoding, bit packing, delta encoding, and run length encoding
      https://en.wikipedia.org/wiki/Dictionary_coder
      https://en.wikipedia.org/wiki/Run-length_encoding
  - *Predicate pushdown*: query optimization technique that filters data at the storage level before retrieving it
  - Optimized for Hive
  - Spark can access ORC as data source
    https://spark.apache.org/docs/latest/sql-data-sources-orc.html

Comparative analysis: https://www.linkedin.com/pulse/comparative-analysis-avro-parquet-orc-understanding-differences-bose

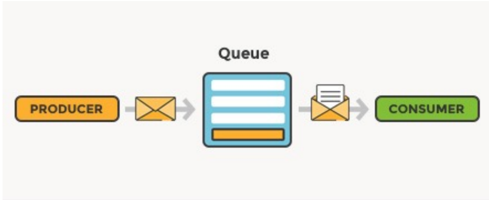
Messaging layer: use cases

- Mainly used in data pipelines for data ingestion or aggregation
- Typically used at the beginning or end of a data pipeline
  - E.g., at beginning of data pipeline:
    - Data from various sensors: ingest data into streaming system for real-time analytics or distributed file system for batch analytics

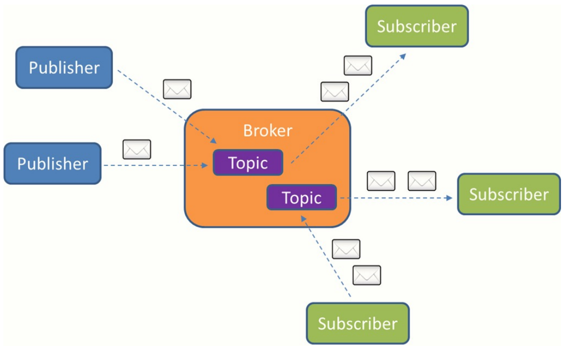


Messaging layer: architectural choices

- Message queue
  - ActiveMQ https://activemq.apache.org
  - RabbitMQ https://www.rabbitmq.com
  - ZeroMQ https://zeromq.org
  - Amazon SQS https://aws.amazon.com/sqs



- Publish/subscribe
  - 👉 **Kafka**
  - 👉 **Apache Pulsar**
  - NATS https://nats.io
  - Redis

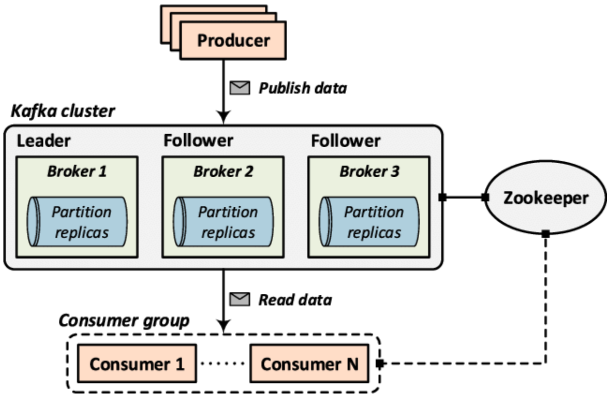


Apache Kafka

- Analyzed in SDCC course
http://www.ce.uniroma2.it/courses/sdcc2425/slides/DS_Communication2.pdf
- In a nutshell
  - Open-source, distributed pub/sub and event streaming platform
  - Designed as a replicated, distributed, persistent commit log
  - Clients produce or consume events directly to/from a cluster of brokers, which read/write events durably to local file system and automatically replicate the events synchronously or asynchronously within the cluster for fault tolerance and high availability
- Let’s recall the main points

Kafka: architecture

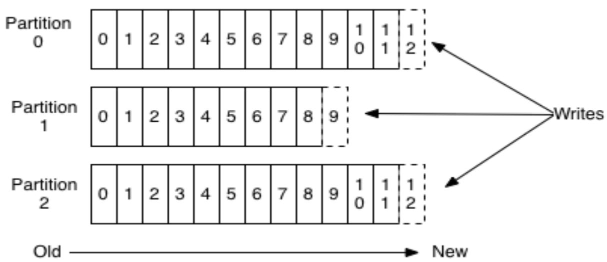
- Kafka maintains feeds of messages in categories called topics
- Producers publish messages to a topic, while consumers subscribe to topics and process published messages



- Kafka cluster: distributed and replicated commit log of data over servers known as *brokers*

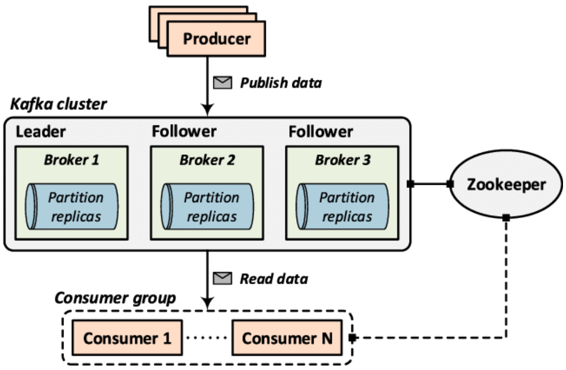
Kafka: topics and partitions

- For each topic, Kafka cluster maintains a partitioned log: topic is split into a fixed number of partitions
- Each partition is an ordered, numbered, immutable sequence of records that is continually appended to
- Each partition is replicated for fault tolerance across a configurable number of brokers
- Partitions are distributed across brokers for scalability



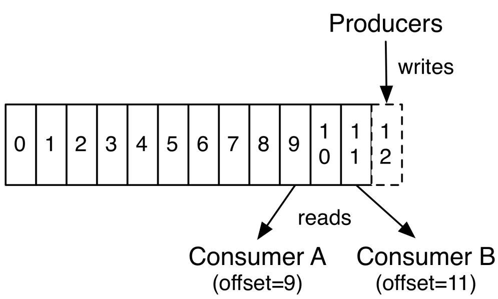
Kafka: partition replication

- Each partition has one leader broker and 0 or more followers
- Leader handles read and write requests
- A follower replicates leader and acts as backup
- Each broker is a leader for some of its partitions and a follower for others to distribute load



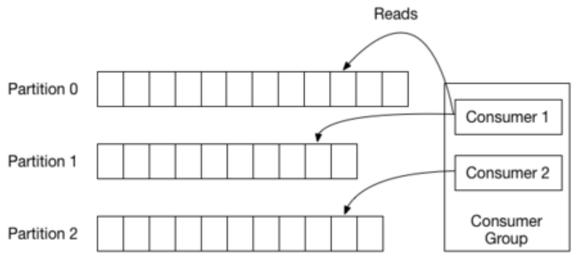
Kafka: partitions

- Producers publish their records to partitions of a topic (round-robin or partitioned by keys), and consumers consume published records of that topic
- Each record is associated with a monotonically increasing sequence number, called offset
  - Kafka provides topic `__consumer_offsets` to store offsets
- Consumers must manage their offset



Kafka: consumers

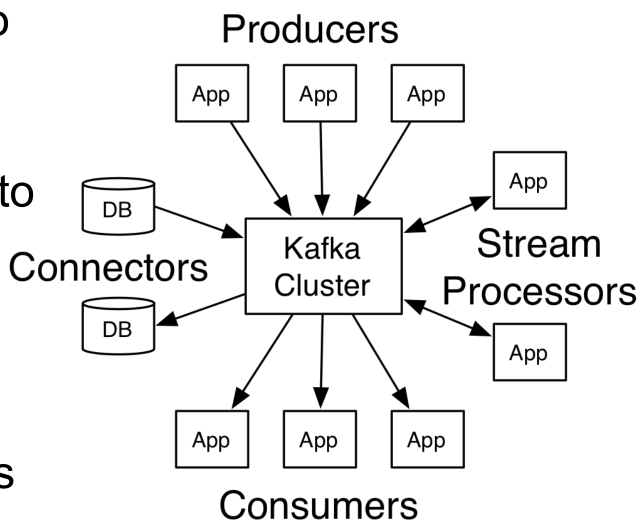
- In Kafka design, pull approach for consumers
  https://kafka.apache.org/documentation.html#design_pull
- Consumers use offset to track which messages have been consumed
  – Replay messages using offset
- Consumers can be grouped into a Consumer Group: a set of consumers sharing a common group ID
  – Group maps to logical subscriber
  – Multiple consumers in a group to increase scalability and fault tolerance



Kafka: APIs

- Core APIs https://kafka.apache.org/documentation/#api

1. Producer API: allows apps to publish records of data (e.g., logs, IoT) to topics
2. Consumer API: allows apps to read records from topics
3. Connect API: reusable connectors (producers or consumers) that connect topics to existing applications or data systems so to move large collections of data into and out of Kafka
   – Connectors for AWS S3, HDFS, RabbitMQ, MySQL, Postgres, AWS Lambda, MongoDB, Twitter, …

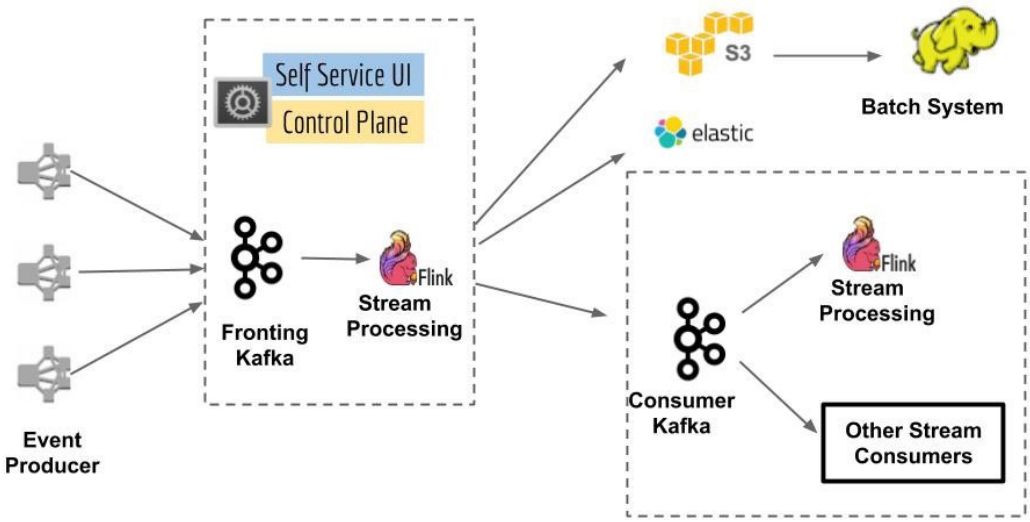


Kafka: APIs

- Streams API: allows transforming streams of data from input topics to output topics
  – Kafka as real-time streaming platform
- Hands-on: use Kafka Streams to process data in pipelines consisting of multiple stages
  https://kafka.apache.org/documentation/streams

Kafka @ Netflix

- Netflix uses Kafka for data collection and buffering

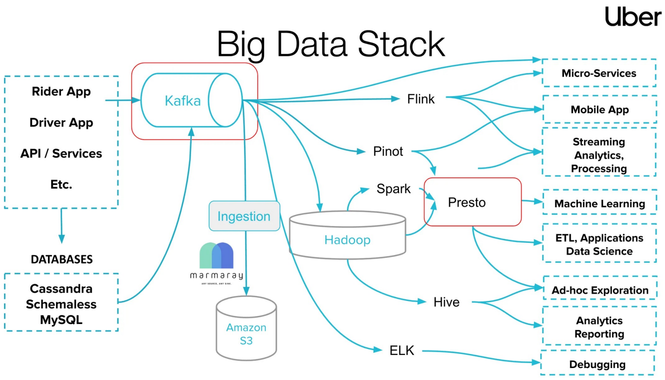


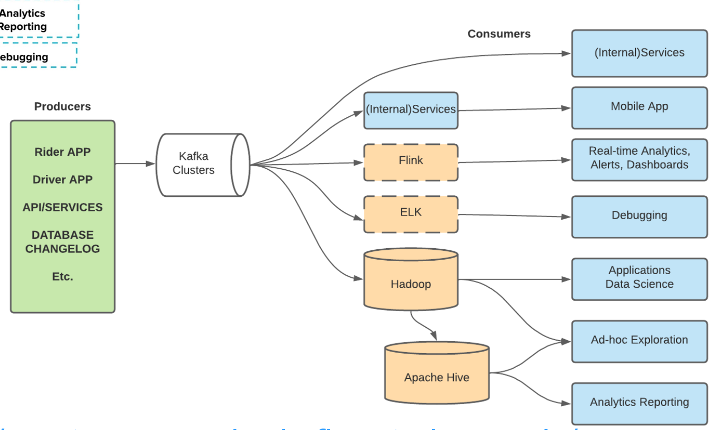
See https://netflixtechblog.com/kafka-inside-keystone-pipeline-dd5aeabaf6bb

Another example: https://www.confluent.io/blog/how-kafka-is-used-by-netflix

Kafka @ Uber

- Uber has one of the largest Kafka deployments

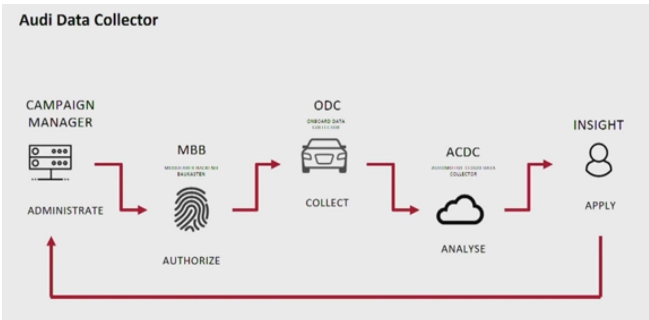


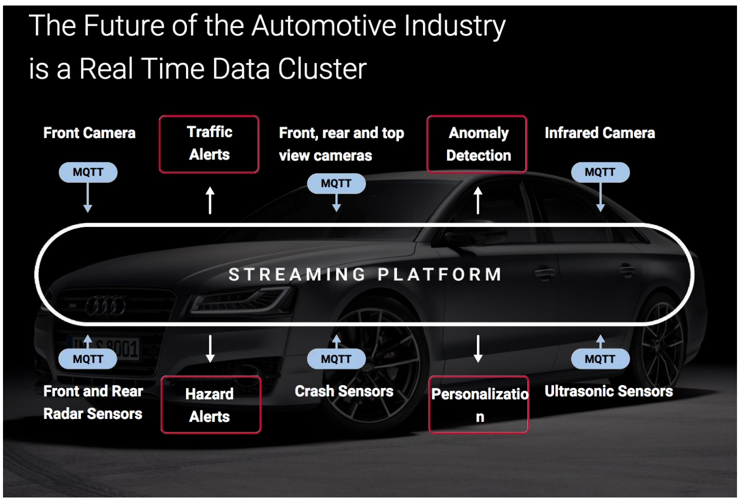


https://www.uber.com/en-IT/blog/presto-on-apache-kafka-at-uber-scale/

Kafka @ Audi

- Audi uses Kafka for real-time data processing
  – 850 sensors in each car



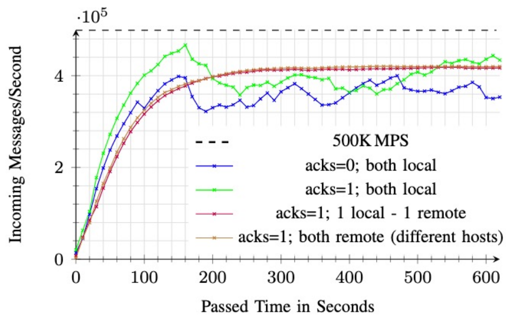


https://www.youtube.com/watch?v=yGLKi3TMJv8

Kafka performance

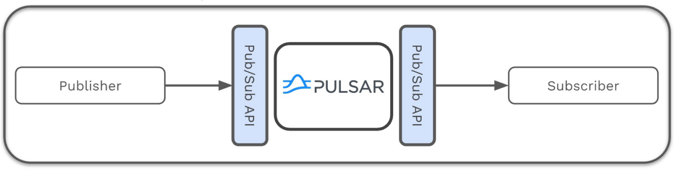
- Performance evaluation study of Apache Kafka

  How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka, ICPADS'20
  https://arxiv.org/pdf/2003.06452
  – Achieves ingestion rate of 421K messages/second or 92 MB/s (single topic with 1 partition and replication factor of 1) on commodity hardware and using 2 senders
  – Ack level choice influences performance: configurations with enabled acks showed better performance



Apache Pulsar

- Cloud-native, distributed messaging and streaming platform, originally developed by Yahoo
  https://pulsar.apache.org

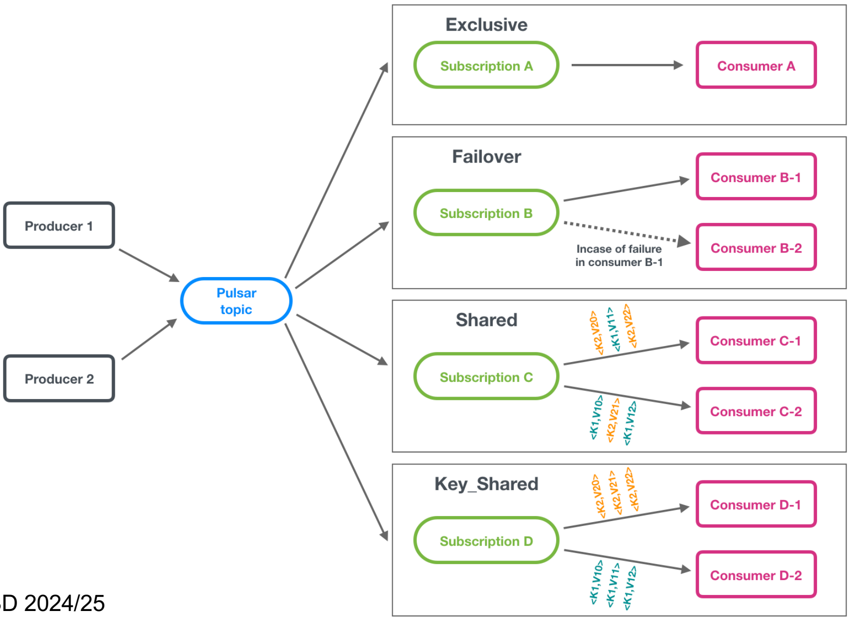


- Scalable, low-latency and durable messaging based on pub-sub pattern, with support for geo-replication
- Multiple subscription types for topics
- Guaranteed message delivery with persistent message storage provided by Apache BookKeeper
- Enables also stream-native data processing through a serverless lightweight computing framework, named Pulsar Functions

  https://pulsar.apache.org/docs/4.0.x/functions-overview

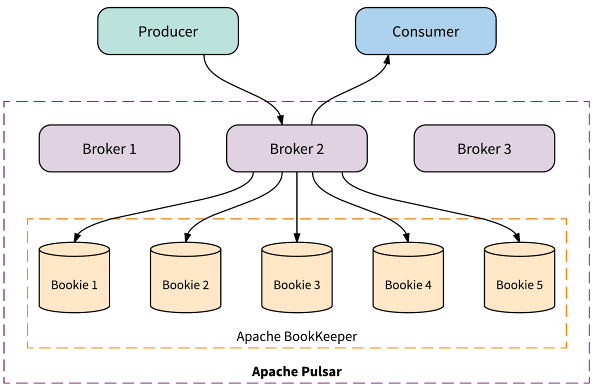
Pulsar: subscription types

- A subscription is a configuration rule that determines how messages are delivered to consumers
- Multiple subscription types: exclusive, shared (or round-robin), failover, and key-shared



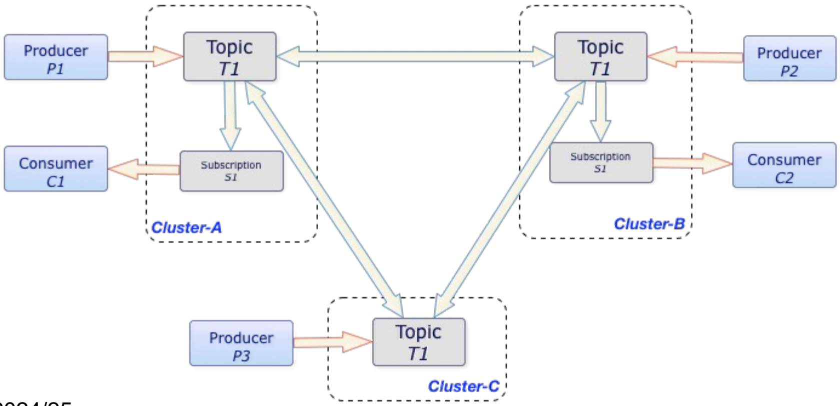
Pulsar: architecture

- Layered architecture designed to provide scalability and flexibility
  - Stateless serving layer and stateful persistence layer
  - Serving layer comprised of **brokers** that receive and deliver messages
  - Persistence layer comprised of Apache BookKeeper storage nodes called **bookies** that durably store messages
    - BookKeeper is a distributed write-ahead log https://bookkeeper.apache.org



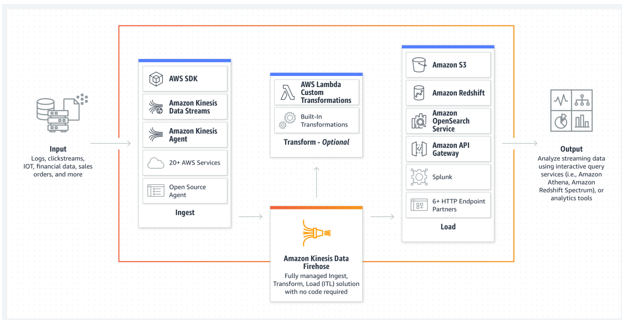
Pulsar: architecture

- Pulsar instance of Pulsar composed of one or more Pulsar clusters
  - Clusters may be geographically distributed and data can be geo-replicated among different clusters
  - Each cluster consiste of one or more brokers, an ensemble of bookies, and a ZooKeeper quorum
  - ZooKeeper is used for cluster-level configuration and coordination



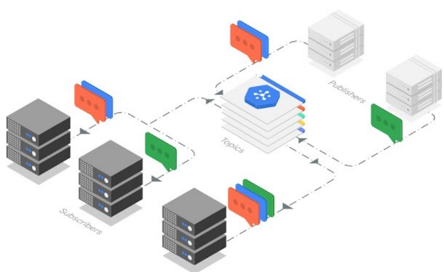
Cloud services for data ingestion

- Amazon Data Firehose
  https://aws.amazon.com/it/firehose
  - Fully managed Cloud service to ingest, transform, and load real-time streams into data lakes (e.g., S3), warehouses, and analytics services



  - Can transform and compress streaming data before storing it
  - Can invoke Lambda functions to transform source data

- Google Cloud Pub/Sub
  https://cloud.google.com/pubsub
  - Fully-managed real-time pub/sub messaging service



Putting all together

- How to schedule and orchestrate data pipelines and workflows?
- Alternatives, including:
  - Apache NiFi
  - Apache Oozie
    - Designed for Hadoop, dated
  - 👉Apache Airflow

Apache Airflow

Apache Airflow

- Open-source platform for developing, scheduling, and monitoring batch-oriented workflows https://airflow.apache.org
  - Initially developed by Airbnb
- Allows users to define *workflows as code*, making them easy to manage, version, and share
  - Workflows are defined in Python
  - Dynamic: workflows are defined in code, enabling dynamic workflow generation, scheduling and parameterization
  - Extensible: wide range of built-in operators and can be extended
  - Flexible: leverages Jinja templating engine, allowing rich customizations https://jinja.palletsprojects.com/en/stable/
- Workflows are represented as DAGs
  - When to schedule workflow execution, how workflow is composed (tasks and their dependencies), callbacks

Apache Airflow: simple example

- A simple DAG
  - When to schedule
  - Two tasks: `BashOperator` and Python function
  - Dependency between the two tasks

```python
from datetime import datetime

from airflow.sdk import DAG, task
from airflow.providers.standard.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```


Airflow and other frameworks

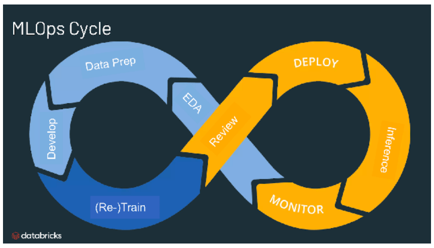
- Can be easily integrated with Spark to schedule and orchestrate Spark jobs alongside other tasks (e.g., data ingestion, validation)
  - Built-in `SparkSubmitOperator` to submit Spark job (e.g., JAR, Python script) to a Spark cluster directly from an Airflow DAG https://airflow.apache.org/docs/apache-airflow-providers-apache-spark/stable/operators.html

```python
submit_job = SparkSubmitOperator(
    application="${SPARK_HOME}/examples/src/main/python/pi.py", task_id="submit_job"
)
```

- Can be also integrated with NiFi
  - From Airflow to NiFi: trigger NiFi Flow from Airflow by sending a POST request to NiFi's REST API
  - From NiFi to Airflow: use NiFi to trigger Airflow DAGs
  - Alternatively, use a message queue or pub/sub system by sending a message from Airflow to NiFi (from NiFi to Airflow)

Airflow: use cases

- Wide range of use cases https://airflow.apache.org/use-cases
  - Automate ETL and ELT pipelines
    - Extract, transform, and load (or extract, load, and transform) data without manual intervention
    - Including: scheduling the data pipeline, handling errors, monitoring, transforming data
  - Manage business operations
  - Infrastructure management, e.g., a Spark cluster
  - Orchestrate MLOps
    - MLOps (ML operations) automates the ML pipeline, from data collection and model training to model deployment and monitoring, and model retraining
    - Can be specialized to generative AI (FMOps and LLMOps)



References

- Apache NiFi documentation https://nifi.apache.org/docs.html
- Apache Kafka documentation https://kafka.apache.org/documentation/
- Apache Pulsar documentation https://pulsar.apache.org/docs/4.0.x/concepts-overview/
  - How to run https://pulsar.apache.org/docs/4.0.x/getting-started-standalone
- Apache Airflow documentation https://airflow.apache.org/docs/