Big Data Ecosystem project

Architecture/design project

Real-time Air Quality Monitoring System

Auteurs:

Norith	UNG
Jade	HARAUCOURT

Group:

Big Data and IA Group 2

Teacher:

M. Yanis BARITEAU

We certify that this work is original, that it is the fruit of a joint effort by the group and that it written independently.

Paris, 01/03/2025

Table of contents

I. Introduction

1.1 Contexte and Project Objectives

Air pollution represents a major environmental and public health challenge, particularly in densely populated urban areas. Emissions from industrial activities, transportation, and other anthropogenic sources contribute to the degradation of air quality, leading to adverse health effects for residents and exacerbating the impacts of climate change. In this context, it is imperative to have an effective air quality monitoring system that provides real-time data, enabling decision-makers to act swiftly to mitigate the negative impacts of pollution.

The objective of this project is to design an innovative platform based on distributed systems, capable of collecting, processing, analyzing, and presenting real-time air quality data. This system aims not only to monitor the most polluted areas but also to anticipate future pollution levels through modeling and forecasting tools. The generated information will be accessible via an interactive dashboard, designed to meet the specific needs of local authorities and the public, guiding concrete actions to improve air quality.

Given the critical nature of air quality monitoring and its potential to address pressing environmental and health challenges, this topic was chosen for its relevance and impact, as well as its ability to showcase the power and scalability of distributed system architectures.

1.2 Importance of Air Quality Monitoring

Real-time air quality monitoring offers several significant advantages:

- **Public Health Improvement:** A better understanding of high-risk areas helps limit the population's exposure to dangerous pollution levels.
- **Rapid Decision-Making:** Local authorities can respond promptly to critical threshold breaches, safeguarding citizens' health.
- **Long-Term Prevention:** Air pollution forecasts facilitate the planning and implementation of structural measures to reduce emissions.
- **Public Awareness:** Access to clear and precise air quality information enhances citizen engagement in environmental actions.

II. Detailed Use Case Description

2.1 Air Quality Data Collection

Collecting air quality data is the first fundamental step to assess and analyze air pollution levels in real time. It is based on installing sensors at strategic locations in the world, to cover areas with differentiated profiles and to study the variation of pollution levels.

- **City center:** High traffic density and economic activity make this area a significant source of pollution.
- **Motorways and peri-urban areas:** These areas experience heavy vehicle traffic, contributing substantially to pollution levels.
- **Industrial Areas:** Specific pollutants are emitted by industries, necessitating focused monitoring in these zones.
- **Residential Areas:** Monitoring here helps study people's exposure to pollution based on their proximity to or distance from polluting sources.
- **Forests and Rural Areas:** These regions serve as a comparison with high-pollution zones, allowing us to observe if pollution impacts green spaces despite low population density.

To evaluate air quality, sensors measure a set of parameters.

Fine particles (PM2.5 and PM10): PM2.5 are suspended particles with a diameter of less than 2.5 µm. They are emitted by vehicles, industries and biomass combustion. PM10 are particles with a diameter of less than 10 µm. They are emitted by sources like PM2.5, but larger. These particles can penetrate deep into the lungs, causing respiratory and cardiovascular problems.

Ozone (O3): It is a gas formed through chemical reactions between nitrogen oxides (NOx) and volatile organic compounds (VOCs) under the action of sunlight. It is an irritant gas which can cause respiratory problems, particularly in vulnerable populations such as individuals with asthma.

Nitrogen Dioxide (NO2): It is created by the combustion of fossil fuels in vehicles and power plants. The consequences of excessive NO2 exposure are respiratory problems, particularly for people living near roads.

Sulfur Dioxide (SO2): It is derived from the combustion of sulfur-containing fossil fuels like coal and oil. This gas irritates lungs and eyes.

Carbon Monoxide (CO): It results from the incomplete fossil fuel combustion. This gas binds with hemoglobin in the blood, reducing oxygen transport.

Sensors transmit the collected data in real time to a centralized infrastructure. Acting as Kafka producers, these sensors continuously send messages to a central Kafka cluster. Kafka serves as a robust distributed messaging system capable of managing large volumes of data in real time.

2.2 Data Analysis

Data analysis enables us to identify areas of high pollution, identify trends and generate actionable insights. The data analysis process is called ETL (Extract, Transform, Load).

In the first stage, raw data are cleaned to eliminate noise and outliers, ensuring accuracy and reliability. Secondly, the data are aggregated by geographical area and time interval to provide an overview of air quality across regions and time periods. Thirdly, abnormal values are identified to determine pollution peaks and highlight areas with significant pollution events. Fourthly, time series data are analyzed to reveal long-term trends, seasonal variations and dependencies on

meteorological conditions, traffic patterns and other factors. Finally, the relationships between pollution parameters, meteorological conditions and human activities are explored to understand the underlying causes of pollution patterns.

Apache Spark enables real-time data processing and analysis, facilitating rapid identification of trends and anomalies. With HDFS (Hadoop Distributed File System), it offers long-term storage for large datasets and supports ETL processes.

2.3 Prediction of pollution levels

To predict pollution levels accurately, we use artificial intelligence (AI) by developing and training predictive models using advanced machine learning techniques. This process involves multiple stages to ensure precise and reliable forecasts.

Before training the models, data must be meticulously prepared. This includes cleaning the raw data to remove noise, handling missing values, and transforming the data into a format suitable for analysis. We then identify and select the most pertinent features for prediction, such as concentration levels of pollutants, meteorological factors, temporal attributes, geographical location... They ensure that the models capture all the critical factors influencing pollution levels.

Once the relevant data is prepared, we begin developing machine learning models. Common algorithms used include Random Forest, Support Vector Machines (SVM), Gradient Boosting Algorithms, Neural Networks... They are trained on historical data, using pollution levels from the past. The goal of the training is that the models learn to associate input features with corresponding pollution levels to improve their accuracy.

Once the training is done, the models are evaluated. Performance metrics are used to assess their accuracy.

Once the models achieve a satisfactory level of accuracy, they are deployed into production systems. The deployment process involves integrating the models into a real-time prediction pipeline. Thus, the models can make efficient predictions.

2.4 Presentation of Results

The results of the data analysis and predictions are presented through an interactive dashboard. This dashboard provides a clear, accurate, and actionable overview of air quality in real time, enabling informed and rapid actions to address environmental challenges.

The dashboard uses powerful visualization tools such as Kibana to represent data in an intuitive way. All important information, such as pollutant concentrations, is updated in real time thanks to the Kafka Cluster. Several types of graphics are implied:

Interactive Maps: They show pollution levels in different areas of the world, allowing users to zoom in and out for detailed information. Moreover, they also display contextual information, such as potential pollution sources, current weather conditions, and historical data, providing a comprehensive understanding of the situation.

Charts and Trends: Lines are included in the dashboard to visualize how pollution levels have changed over time and to get the trends. The users can select different parameters such as the pollutants, time periods and geographic areas. Charts make comparisons with previous year data to analyze the effectiveness of pollution reduction measures.

Alerts and Notifications: The automated alerts and notifications indicate when pollution levels are high or when future peaks are predicted to adopt better behaviors.

Reports: Detailed reports about air quality also include further analysis. They feature daily, weekly, and monthly summaries, and incorporate statistical measures. The reports are composed of detailed technical documents for scientists and summaries for policymakers.

III. Presentation of the Technologies involved

In this section, we present the technologies selected for the design of the air quality monitoring system. Each technology is analyzed in detail, highlighting its objectives, its specific advantages, the reasons for its choice, and the alternatives considered.

3.1 Apache Kafka

Apache Kafka is an open source distributed streaming platform designed to handle real-time data streams. Written in Scala and Java, it runs on the JVM (Java Virtual Machine). Kafka is capable of transporting large amounts of data quickly and efficiently. It is also used to create data pipelines and streaming applications. The main components of Kafka are:

- **Producers**: These are the applications that send data to Kafka. In our project, this role is played by sensors.
- **Topics**: These are the message streams sent by producers. Topics are used to organize data.
- **Consumers**: These are the applications or services that read data from topics.
- **Brokers**: These are the nodes in the cluster that store data and respond to requests from producers and consumers.

Here are the main criteria that led to this choice:

- **Scalability**: Kafka can easily scale to handle hundreds/thousands of data streams while maintaining low latency.
- **Reliability**: Data is replicated across multiple brokers which ensure availability even if there is a failure of the event.

- **Performance**: Thanks to its distributed architecture and replication capability, Kafka offers high performance.
- **Integration**: Kafka easily integrates with tools like Apache Spark or Hadoop, making it perfect for building complete architecture.

Specific benefits

- **Real-time processing**: Kafka transports air quality data in real time, allowing for rapid response to pollution spikes.
- **Fault tolerance**: If a broker fails, the data remains available thanks to replication. This ensures that the data flow does not stop.
- **Scalability**: As our sensor network grows, Kafka can handle the increased data volume without any issues.
- **Integration with Spark**: It can easily be integrated with Apache Spark to process data in real time and build models to predict pollution levels.

Alternatives Considered

Before choosing Apache Kafka, we took into consideration two other options. **Apache Pulsar** is a distributed messaging system that offers similar features to Kafka, such as replication and fault tolerance. Nevertheless, Kafka integrates better with the other technologies in our project compared to Pulsar which has more difficulties. The other alternative is **RabbitMQ** which is a reliable open-source message broker, ideal for simple applications. The inconvenients are that it is less performant and suited to processing large amounts of data in real time.

3.2 Apache Hadoop

Hadoop is a stack of Open-Source software offering all the functionalities needed to build a Data Lake (repository of data stored in its natural/raw format) and exploit the data stored in it. Most Hadoop projects are maintained by the Apache Software Foundation and can be found on their GitHub. It is mainly built using Java or running in the JVM (Java Virtual Machine) and its preferred execution environment is Linux. Hadoop's goal is to store and process large amounts of distributed data. The Hadoop ecosystem is composed of three main components:

- HDFS (Hadoop Distributed File System): The distributed file system that allows storing large amounts of data in a reliable and scalable manner.
- YARN (Yet Another Resource Negotiator): The cluster manager that manages cluster resources and schedules tasks.
- MapReduce: The execution engine that enables parallel and distributed data processing.

Here are the main criteria that led to this choice:

- **Storage capacity:** Hadoop can store data on hundred/thousands of nodes
- **Data Replication**: Avoid data loss and optimize the access
- **Large File Support:** it can handle files ranging from gigabytes to several terabytes
- **High throughput**: It is designed to provide high throughput, allowing large amounts of data to be processed efficiently

- **Unix-like file system**: It provides a Unix-like file system, making data management and access easier
- **Efficient architecture:** Hadoop's architecture is based on master components that coordinate worker nodes to perform tasks. The master node, known as the NameNode, handles the file system metadata, including the path of the file, block information, and the location of the blocks within the cluster. The worker nodes, known as DataNodes, store the actual data blocks on disks and handle read and write operations. This architecture is efficient because it ensures that data is correctly managed and is never lost.

Specific benefits

- **Scalable Storage:** HDFS can store large amounts of data in a scalable manner, which is essential for our real-time air quality monitoring project because the levels of pollution depend on numerous parameters.
- **Distributed Processing**: MapReduce allows data to be processed in a distributed manager, which increases efficiency and reduces processing time.
- **High Availability:** Data replication across multiple nodes ensures that data is available even if some nodes fail. This is important for our project because we are studying real-time data and need to make accurate predictions. The architecture must work reliably every day and at all times without errors.
- **Easy Integration:** Hadoop easily integrates with other open-source technologies such as Apache Spark and Apache Kafka, making it straightforward to implement our overall architecture. This integration allows for seamless data flow and processing, enhancing the efficiency and effectiveness of our system.

Alternatives considered

As part of this project, several alternatives to Apache Hadoop were considered to meet the needs for storing and processing large amounts of data. Firstly, **Amazon S3**, was studied for its ability to store objects in a scalable and reliable manner. Additionally, this service was taught in school, which made us hesitant to use it. However, we decided to not adopt it for several reasons. Amazon S3 is not an open-source solution, which does not align with our preference for open-source technologies. There are also costs associated which can become very expensive as data volumes increase. It could also be difficult to manage the architecture to interact with the other technologies such as Kafka or Nifi.

Secondly, **Google Cloud Storage** was also considered. Since we studied the cloud in school, we thought it might be a good idea. As Amazon S3, we did not decide to adopt it because of the costs and it is not Open-Source. Moreover, when we use Google Cloud Storage, we have to migrate data from Kafka to this service. This step is difficult.

3.3 Apache Spark

Apache Spark is a fast, distributed, and versatile cluster computing system. It is an open-source project written in Scala and running on the Java Virtual Machine (JVM). Spark is integrated into the Hadoop ecosystem for storage and resource management, and it can work with cluster managers such as Hadoop YARN. Spark is particularly suited for processing large datasets and allows ETL (Extract,

Transform, Load) processes as well as building machine learning models with real-time data streams. In addition, it can process all three types of data: structured, semi-structured, and unstructured.

Apache Spark is mainly composed of four components:

- **Driver Program**: The program manages data processing tasks. It divides tasks into multiple stages before distributing them to cluster nodes.
- **Cluster Manager**: It manages cluster resources and schedules tasks.
- **Executors**: It is a processor which executes data processing tasks on cluster nodes. The executing tasks are assigned by the Driver Program to each executor.
- **RDDs (Resilient Distributed Datasets)**: Spark's fundamental data structures, enabling distributed data storage and reliable manipulation.

Apache Spark was chosen for the following reasons:

- **Speed**: Spark can process data quickly, thanks to its in-memory execution engine that reduces disk I/O.
- **Flexibility**: It supports multiple programming languages (Scala, Java, Python, R) and can be used for a variety of data processing tasks, such as batch processing, real-time processing and machine learning.
- **Fault Tolerance:** To provide fault tolerance, it uses RDDs. In case of a node failure, data is automatically recovered in the event.
- **Integration:** it easily integrates with other open-source technologies such as Hadoop, Kafka, HBase.

Specifics benefits

Apache Spark aligns perfectly with the needs of the air quality monitoring project.

- **Capacity of analysis**: Its skills in Machine Learning enable the development of predictive models to anticipate pollution levels based on historical data and real-time data.
- **Scalability**: Its environment can manage large amounts of data and it is adapted to the evolution of data coming from the Kafka cluster
- **Integration with Hadoop:** The seamless integration with Hadoop simplifies data storage and ETL workflows.

Alternatives considered

In the context of this project, several alternatives to Apache Spark were considered to meet the needs of analysis and creation of predictive models. Among them, **Apache Flink** attracted attention for its ability to store distributed data. Flink is particularly suited for real-time data processing and batch analysis with batch processing. However, Spark offers better integration with other open-source technologies used in our project. Integrating Flink into the overall architecture and creating interconnections between all the components of the project could be more complex.

Another alternative examined is **Apache Storm**. Storm is well suited for applications requiring real-time processing. However, Spark stands out for its flexibility and its ability to handle batch data with batch processing.

In conclusion, although Apache Flink and Apache Storm offer advantages in certain contexts, they do not meet the specific needs of this project as effectively as Apache Spark, especially in terms of flexibility.

3.4 Apache HBase

Apache HBase is a distributed NoSQL database designed to manage large quantities of structured and semi-structured data. It is based on the Google Bigtable architecture and is specially optimized for applications requiring fast, real-time access to massive volumes of data.

HBase is ideal for use cases where data is voluminous, sparse and needs to be accessed quickly, such as in sensor monitoring systems where low latency is crucial.

For the air quality monitoring system, the data collected by the sensors is voluminous and scattered, and requires strict consistency. HBase was chosen for the following reasons:

- **High scalability:** HBase can easily scale up to manage petabytes of data, guaranteeing constant performance even with a significant increase in the number of sensors or data collected.
- **Strict consistency:** Unlike Cassandra, which uses eventual consistency, HBase guarantees strong consistency, which is crucial for guaranteeing the accuracy of the data collected.
- **Compatibility with Hadoop:** Native integration with the Hadoop ecosystem (HDFS and MapReduce) enables massive analysis of stored data.
- **Low latency:** Designed for fast reads/writes, HBase is suited to scenarios where data needs to be accessible in near real time.

Specific benefits

- **Distributed, fault-tolerant architecture:** Data is distributed across multiple nodes, ensuring availability in the event of hardware failure.
- **Flexible data model:** Able to manage semi-structured data from heterogeneous sensors.
- **Optimized performance for random reads/writes:** Ideal for frequent and irregular queries on large datasets.

Alternatives considered

As part of this project, several alternatives to Apache HBase were considered to meet the needs for storing and managing the large, sparse data from sensors. Among them, **Apache Cassandra** attracted attention for its master-master architecture, which guarantees high availability and exceptional fault tolerance. However, this architecture favors eventual consistency, which can lead to temporary inconsistencies in the data. For an application requiring strict consistency, such as a real-time monitoring system, this limitation is critical. In addition, although Cassandra excels at massive write loads, its read performance, particularly for sparse and large data, is inferior to that of HBase.

Another alternative, **MongoDB**, was examined for its flexibility in managing semi-structured data. MongoDB is well suited to scenarios where data models may evolve or be dynamic, but it is not optimized for massive volumes of data requiring fast, consistent access. In a context where frequent and intensive queries on very large databases are required, MongoDB becomes less efficient than HBase.

In conclusion, although Cassandra and MongoDB offer advantages in certain contexts, they do not meet the specific needs of this project as effectively as HBase, particularly in terms of consistency, latency and compatibility with the Hadoop ecosystem.

3.5 Apache NiFi

Apache NiFi is a real-time data flow management platform. It offers an intuitive graphical user interface that enables complex data flows to be designed, monitored and managed visually. NiFi is specially designed to orchestrate heterogeneous data flows and integrate them easily with other systems.

NiFi was chosen for its ability to manage complex data flows while ensuring high reliability. Here are the main criteria that led to this choice:

- **Flexibility:** NiFi can manage data flows from multiple sensors in a variety of formats (JSON, CSV, etc.).
- **Powerful orchestration:** Allows data to be transformed, filtered and routed to different systems, such as HBase or Elasticsearch.
- **Reliability:** Error management and failover features ensure continuity of critical data flows.
- **Intuitive user interface**: Reduces technical complexity and facilitates data flow maintenance.

Specific benefits

- **Real-time monitoring:** Allows you to monitor the status of data flows and intervene quickly in the event of a problem.
- **Integrated security:** Includes encryption, access control and authentication functions.
- **Horizontal scalability:** Can be extended to manage a greater number of data flows or volumes.

Alternatives considered

For data flow management and orchestration, **Apache Flume** has been considered as a serious alternative to Apache NiFi. Flume is designed for ingesting large quantities of data to storage systems such as HDFS. However, it lacks the flexibility to handle complex data flows involving advanced transformations or conditional routing. In addition, Flume does not offer a graphical interface for workflow management, which can make it more complex to configure and maintain.

Another tool under consideration is **Talend**, a powerful ETL solution with advanced data management features. Talend offers great versatility and is used in a variety of sectors for data integration and transformation. However, its configuration is more technical and demanding, and it is primarily suited to batch processing rather than real-time workflows. This makes it less relevant for a system requiring continuous and dynamic orchestration of data from real-time sensors.

So, while Flume and Talend are viable solutions for other types of projects, they don't match up as well as NiFi to the air quality monitoring system's requirements for flexibility, ease of management and support for real-time streams.

3.6 Elasticsearch

Elasticsearch is a distributed search and analysis engine designed to manage and explore large volumes of data in real time. Its rapid indexing capabilities and advanced search and aggregation features make it an essential tool for systems requiring high-performance analysis.

For the interactive dashboard of the air quality monitoring system, Elasticsearch is particularly well suited thanks to :

- **Fast indexing :** Collected data is immediately indexed for near real-time access.
- **Advanced search:** Allows full-text searches, aggregations and complex filters to analyze data.
- **Integration with visualization tools**: Works perfectly with Kibana to create interactive, dynamic dashboards.

Specific benefits

- **Horizontal scalability:** Can handle a growing load without loss of performance.
- **Fast response time:** guarantees rapid queries on large datasets.
- **Schema flexibility:** Supports different data formats from multiple sources.

Alternatives considered

For real-time search and analysis needs, two main alternatives to Elasticsearch were explored: **Apache Solr** and **Splunk**.

Apache Solr, like Elasticsearch, is based on the Lucene library, and both tools offer similar full-text search and aggregation capabilities. However, Solr has historically been geared towards more static or less frequent search scenarios. For this project, where data needs to be indexed and searched in real time, Elasticsearch performs better thanks to its native distributed architecture and optimizations for continuous flow scenarios.

On the other hand, Splunk is a well-established commercial solution, known for its powerful analytics and advanced data visualization capabilities. However, Splunk is a proprietary solution, with high acquisition and maintenance costs, making it less attractive for a project looking to leverage open-source solutions. In addition, its complexity of deployment and its focus on large enterprises add constraints that are not suited to this project.

To sum up, while Solr and Splunk each bring strengths in their respective areas, they do not meet the specific needs of real-time search, scalability and easy integration with visualization tools such as Kibana as well as Elasticsearch does.

Conclusion

The alternatives examined, although relevant in other contexts, do not meet the requirements of this project as precisely as the technologies selected. Kafka, Hadoop, Spark, HBase, NiFi and Elasticsearch are the most appropriate solutions for guaranteeing a high-performance, reliable and scalable architecture that meets the challenges of a real-time air quality monitoring system.

To clearly summarize the rationale behind the selection of each technology, the table below compares the chosen solutions with the alternatives considered, highlighting the key reasons for each choice.

Criterion	Selected Technology	Considered Alternatives	Reason for Selection
Real-time data	**Apache Kafka**	Apache Pulsar	Better integration with the other technologies
		RabbitMQ	More performant, more adapted to manage large amounts of data in real time
Data Storage and resource management	**Apache Hadoop**	Amazon S3	Open-source, better scalability, none costs for large-scale data storage and architecture master worker efficient
		Google Cloud Storage	Open-source, better integration with the other technologies and architecture master worker efficient
Data Processing	**Apache Spark**	Apache Flink	More flexibility for real-time and batch processing
		Apache Storm	Gives better support for machine learning and data analytics
Data Storage	**Apache HBase**	Apache Cassandra	Strict consistency and better-read performance for large, sparse datasets
		MongoDB	Superior performance for large-scale read/write operations and integration with Hadoop
Data Flow Orchestration	**Apache NiFi**	Apache Flume	Intuitive interface, flexibility for complex transformations, and real-time support
		Talend	Simpler to manage and better suited for real-time streams compared to batch processing
Analysis and Visualization	**Elasticsearch**	Apache Solr	Superior performance for real-time search and seamless integration with Kibana
		Splunk	Lower cost and preference for open-source technologies offering greater flexibility

IV. Overall Project Architecture

4.1 Global Architecture Diagram

The overall architecture of the real-time air quality monitoring system is designed to respond effectively to the needs of environmental data collection, processing, analysis and visualization.

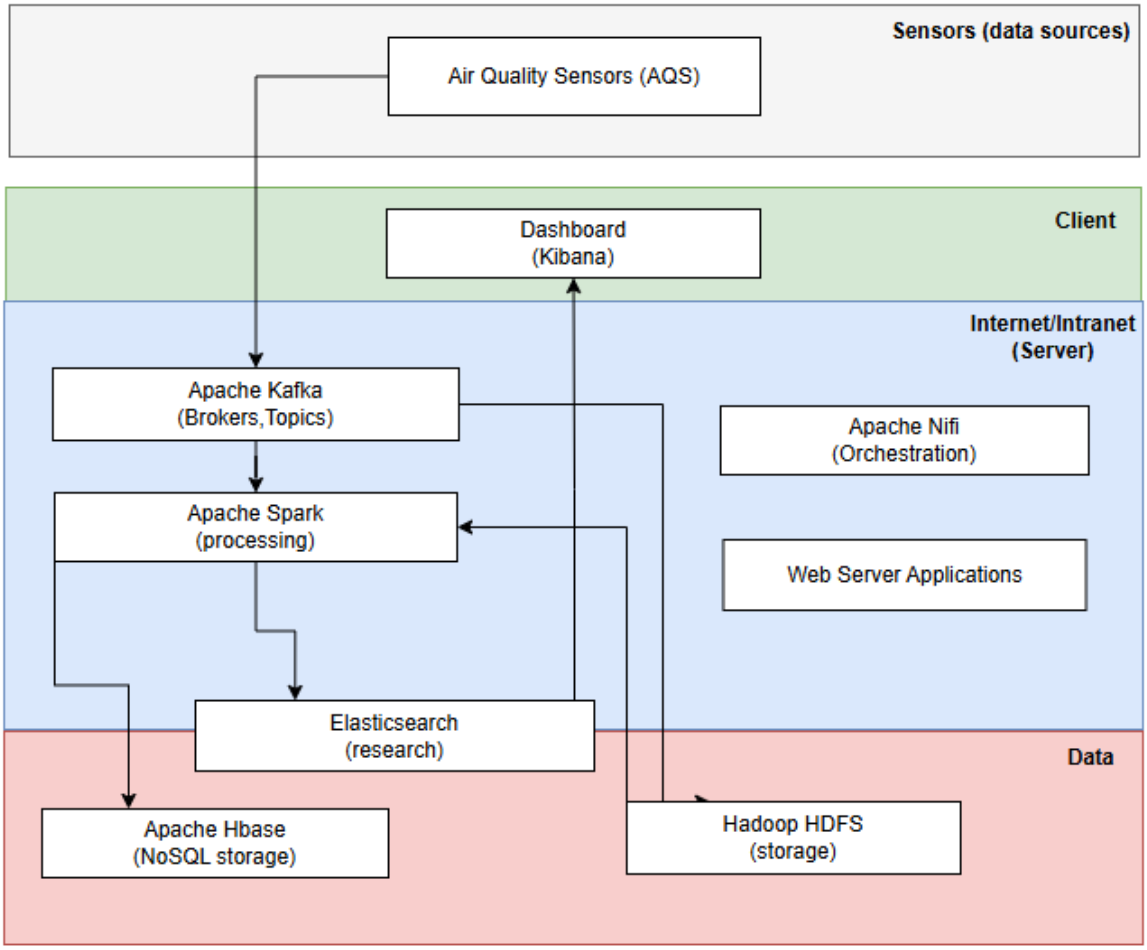

Diagram of the global architecture

This diagram provides an overview of the key components and their interconnections. The architecture is organized into four main layers:

- **Sensors (data sources):** Ensure the collection of environmental measurements in the field.
- **Client:** Provides a user interface for visualization and analysis.
- **Server (Internet/Intranet):** Manages data processing, flow orchestration and communication between the various components.
- **Data:** Includes systems for storing, searching and indexing information.

Each layer interacts seamlessly to provide accurate, real-time information on air quality.

4.2 Architectural elements

Air Quality Sensors (AQS)

Air quality sensors measure various pollutants such as PM2.5, PM10, NO2, SO2, O3 and CO. They are installed in a number of strategic locations around the city, enabling pollution levels to be monitored in real time. These sensors transmit the data collected via standard protocols such as MQTT or HTTP to a centralized system for further processing.

Example: The SDS011 sensor is used to measure fine particles (PM2.5 and PM10), while the MQ-131 detects ozone (O3) concentrations.

Apache Kafka (Brokers, Topics)

Apache Kafka acts as a distributed messaging platform that guarantees the reliable reception of sensor data in real time. The data is divided into 'topics' (specific themes) to enable efficient distribution to the various components of the system. Kafka ensures the scalability and resilience that are essential in a big data architecture.

Example: One topic can be dedicated to PM2.5 measurements and another to CO measurements, ensuring a clear organization for data consumption

Hadoop HDFS (Hadoop Distributed File System)

HDFS is used for long-term storage of raw data. This distributed file system offers massive storage capacity and fault tolerance, essential for managing large volumes of data generated by sensors. HDFS is particularly well suited to architectures that require data persistence for retrospective analysis.

Apache Spark (Data Processing and Analysis)

Apache Spark is the platform of choice for real-time data processing and complex analysis. Spark enables the execution of data transformations, advanced analytical calculations and machine learning algorithms. These analyses include identifying areas of high pollution and predicting air quality trends.

Example: Real-time data is enriched, cleaned and used to predict pollution peaks in certain areas.

Apache HBase (NoSQL database)

Apache HBase is a distributed NoSQL database designed for storing processed data and analysis results. It provides rapid access to semi-structured data, which is essential for powering interactive user interfaces and dashboards. HBase is particularly well suited to managing large quantities of data in near-real time.

Elasticsearch (Data Search and Analysis)

Elasticsearch is a search and analysis solution designed to index processed data. It enables fast, accurate queries to be made, making it easier to analyze and visualize data. Elasticsearch is a key component for powering tools such as Kibana, enabling rapid search and in-depth analysis of the data collected.

Apache NiFi (Data Flow Orchestration)

Apache NiFi manages the orchestration and transfer of data flows between the various components. It also enables data to be transformed and filtered according to requirements, guaranteeing the consistency and synchronization of data flows between sensors, analysis tools and databases.

Example: NiFi can automate the transfer of raw data from Kafka to HDFS while applying format transformations.

Dashboard tools (Kibana)

Kibana is used to visualise real-time air quality data and analysis results. It provides interactive dashboards that allow users to monitor pollution levels, compare historical and real-time data, and identify areas at risk.

Example: A Kibana dashboard can display a map of the city with air quality indicators by zone, accompanied by trend graphs.

Web Server Applications

In addition to generating reports, web server applications can be developed to provide interactive and user-friendly access to air quality data. These applications can offer real-time monitoring, historical data comparison, and predictive analytics through a web interface.

Example: A web application could provide users with real-time air quality alerts and personalized recommendations based on their location.

4.3 Data Flow

The data flow starts with the air quality sensors (AQS), which collect pollution data in real time. This data is sent to Apache Kafka, which acts as a distributed messaging system. Kafka receives the data from the sensors and distributes it to two main destinations: Hadoop HDFS for long-term storage and Apache Spark for real-time processing and analysis.

The raw data stored in Hadoop HDFS is used for further analysis by Apache Spark. Spark processes this data to identify areas of high pollution and predict future pollution levels. The results of these analyses are then stored in Apache HBase, a NoSQL database that provides fast and reliable access to the processed data.

At the same time, the data processed by Apache Spark is indexed in Elasticsearch to enable rapid search and analysis. Elasticsearch facilitates the analysis and visualisation of air quality data in real time.

Apache NiFi plays a crucial role in orchestrating data flows between different sources and destinations. NiFi manages and transforms data flows flexibly and efficiently, ensuring data integration and consistency.

Finally, data from Elasticsearch and Apache HBase is used by web server applications to generate interactive dashboards. These dashboards, such as those created with Kibana, enable users to monitor pollution levels in real time, identify areas at risk and make informed decisions to improve air quality. Additionally, web server applications can provide real-time alerts and personalized recommendations based on user location, enhancing the overall user experience.

V. Interconnections between Technologies

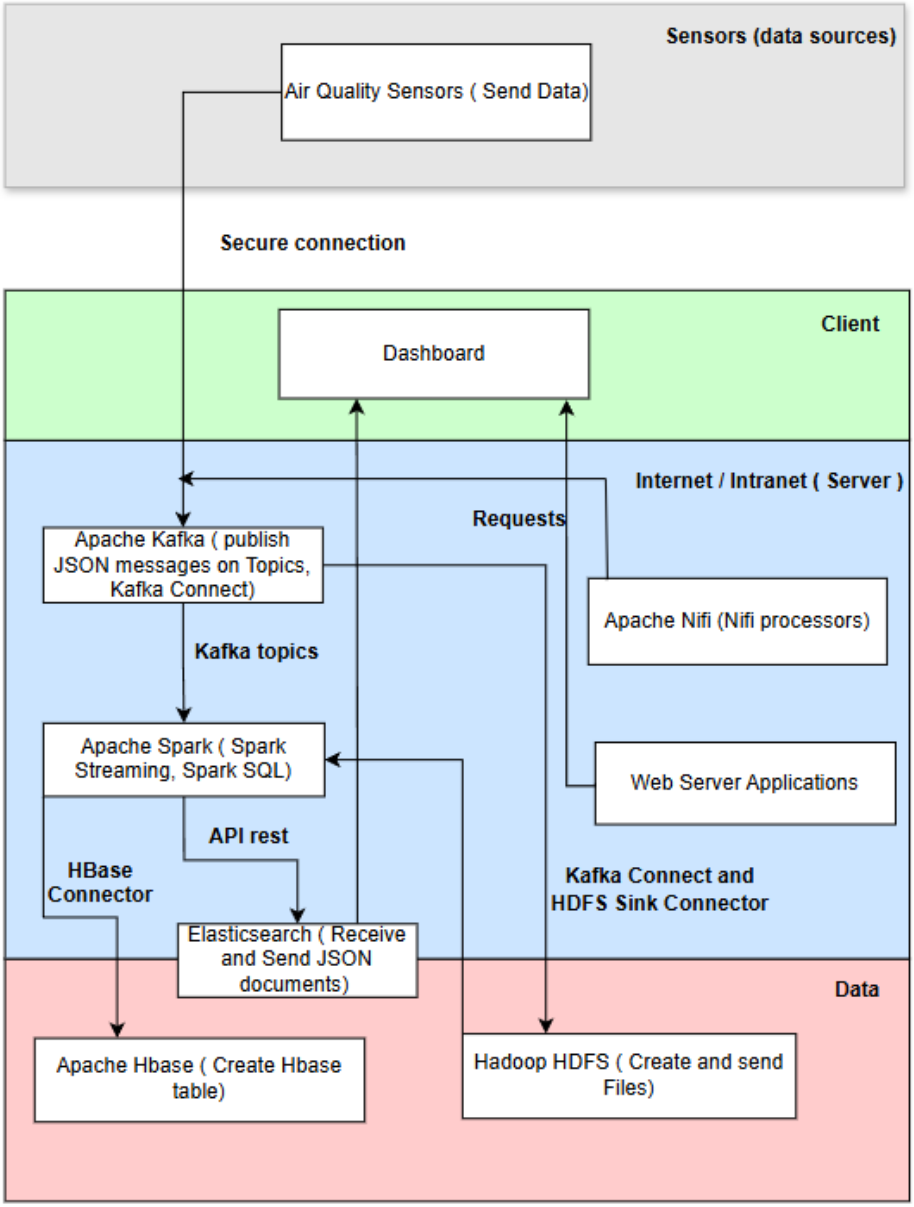

Diagram of the interconnexions between the technologies

This diagram shows how the technologies involved are interconnected.

5.1 Sensors to Apache Kafka

Air quality sensors collect air data in real time. They send it in parallel to a central server (Apache Kafka) via a secure network connection, using network communication protocols (TCP/IP) and computer network security protocols such as TLS (Transport Layer Security). Secure data is sent to Kafka in the form of JSON messages, which are then published on specific Kafka topics. A Kafka topic is a category or stream of messages to which producers publish data, and to which consumers subscribe to read the data. Kafka topics are divided into partitions, and each partition is an immutable sequence of messages. Partitions enable Kafka to parallelize message processing and ensure fault tolerance by replicating partitions across multiple brokers. Topics can be configured for different levels of replication and data retention, depending on the needs of the application.

5.2 Apache Kafka to Apache Spark

Apache Kafka sends data via Kafka topics to Apache Spark. Apache Spark reads data from topics using an extension called Spark Streaming. Spark Streaming processes the continuous data stream in real-time. The operation of this extension is done in two main steps: first, it divides the data stream into micro-batches, and then the Apache Spark processing engine processes these batches. Here are the main steps in which Apache Spark receives topics.

Connecting to Kafka: When receiving topics, they are secured by security protocols (Transport Layer Security) and network communication (TCP/IP). Spark Streaming accesses the received Kafka topics using Kafka configuration parameters, such as broker addresses and topic names.

Reading Process: Once connected, Spark Streaming reads messages in real-time. The messages are read as continuous data streams called DStreams (Discretized Streams). DStreams are abstractions of data streams that allow you to process data continuously.

Data Processing: Spark Streaming applies transformations and actions to data on DStreams. Transformations allow us to filter, map, reduce, and aggregate data. Actions allow us to send the results to external systems, write the results to storage systems, or print the results.

Micro-batch Management: Data is divided into micro-batches to be able to process it efficiently. Apache Spark's batch processing engine processes these micro-batches one by one. Micro-batches allow Spark Streaming to process data in real-time while using Apache Spark's batch processing engine. This approach combines the advantages of real-time and batch processing, providing great flexibility and efficiency.

5.3 Apache Spark to Elasticsearch

Apache Spark processes the data, cleans it, transforms it, and reduces it. Once this step is completed, it sends it to Elasticsearch via a REST API. A REST API (or RESTful API) is an interface that allows computer systems to communicate with each other via the HTTP protocol by respecting the principles defined by the REST (Representational State Transfer) architecture. Apache Spark sends JSON documents to Elasticsearch containing the data that has been processed and cleaned. Here are the steps for receiving and analyzing JSON documents.

Receiving Data: Elasticsearch receives JSON documents via HTTP requests. These requests are sent to a specific Elasticsearch URL, which contains the data to be indexed.

Parsing Documents: Elasticsearch opens JSON documents, reads them, and analyzes their content. It then creates a mapping. A mapping is a plan for each type of document. It indicates for each field of the document the type of data it contains and how this data should be analyzed. For example, a text field can be parsed to be split into individual words (tokenized) and transformed into lowercase. The mappings are then stored in a format called "inverted indexes", specific to Elasticsearch. An inverted index is a data structure that permits us to quickly search for specific values in documents.

Document Storage: Documents are then stored in segments. Each segment contains a portion of the data and information about where the data is located. Segments are immutable files, meaning that once created, they cannot be modified. This property is often referred to as WORM (Write Once, Read Many). This ensures data consistency and better versioning. Segments are periodically merged to optimize storage space and improve search performance.

Search and Retrieval: Once indexed, documents can be searched using queries sent to the Elasticsearch REST API. These queries are executed against the shards of the index. A shard is a partition of the index that contains a portion of the data. Shards allow data to be distributed across multiple nodes in the Elasticsearch cluster, improving scalability and fault tolerance. After a search is performed, the results are aggregated. The results are then returned as JSON documents.

5.4 Apache Spark to Apache HBase

Apache Spark is connected to Apache HBase via an HBase connector. This connector allows Spark to write directly to HBase. Only read and write operations are allowed. When Spark writes to HBase, the data is stored as HBase tables in NoSQL. Each HBase table is divided into regions, each region being a partition of the data. Regions are horizontal partitions of a table. In addition, regions are assigned to multiple nodes in the HBase cluster to ensure fault tolerance and scalability. Data reception and storage takes place in several steps:

Data reception: Data is received as table rows, containing the keys for each row and the values of the columns. Indeed, HBase works on the principle where a key is equal to a value, and each column is stored separately. This data model allows great flexibility and efficient management of sparse data.

Read and Write Process: HBase writes data to the specified tables. They are written to the appropriate regions of HBase. The HBase API update (put) operations which are used to write the data to the regions. Each region contains a portion of the table data and is distributed across multiple nodes in the cluster.

Region Management: Regions, which are distributed across the cluster nodes, are used to perform read and write operations and provide fault tolerance.

5.5 Apache Kafka to Hadoop HDFS

To connect Apache Kafka to Hadoop HDFS, we use Kafka Connect and the HDFS Sink Connector.

Kafka Connect: It is a framework which connects to external systems, such as databases, storage systems, and applications. In this case, Hadoop HDFS is the data storage system.

HDFS Sink Connector: It is a Kafka connector which transfers data from Kafka to HDFS.

Different Steps

The first step is to connect with the HDFS Sink Connector parameters, such as Kafka broker addresses, topic names, storage paths in HDFS, data format, and serialization options.

Kafka sends messages to specified topics. Kafka Connect is able to read these messages using Kafka consumers. These messages are read as continuous data streams.

Subsequently, the data is transformed by filtering, mapping, and format conversion operations (e.g., from JSON to Avro).

Next, Kafka Connect writes the transformed data to HDFS using the HDFS Sink Connector. The data is written as files to the specified storage paths in HDFS so that the data can be stored in the long term. These files are organized temporally (by date and time).

Finally, the files are divided into blocks, with each block replicated across multiple nodes in the cluster to ensure fault tolerance. A block has a fixed storage unit, 128 MB by default, allowing HDFS to perform read and write tasks in parallel.

5.6 Hadoop HDFS to Apache Spark

Hadoop HDFS stores data in clusters of machines and Spark performs fast and interactive analyses on large data. To process data stored in HDFS with Apache Spark, several steps are required:

Configuration

First, Apache Spark must be configured so that it can read data directly from HDFS. It is necessary to specify the storage paths of files in HDFS, the addresses of HDFS nodes and Spark-specific configurations such as allocated memory.

Reading Data

To read data directly from HDFS, Apache Spark uses programming interfaces. The data read is in the form of files containing messages stored in HDFS and is stored in DataFrames.

DataFrame: Distributed and immutable data structures representing data tables. They allow data to be manipulated and transformed.

Data Transformation

After reading the data, Apache applies transformations to the DataFrames to filter, map, reduce and aggregate the data and triggers actions.

SQL Query Execution

Once the DataFrames are ready. It is possible to execute SQL queries to analyze the data. The results returned are in the form of DataFrames.

Writing the Results

The results can be written to external storage systems like HDFS or HBase.

The writing can be done in HDFS in the form of files to be able to store the data in the long term.

The data can also be written in HBase tables for fast access and efficient management of unstructured data. HBase is a distributed NoSQL database that offers fast read/write access for sparse or semi-structured data.

5.7 Elasticsearch to Dashboard

After being sent by Apache Spark to Elasticsearch, the data has been indexed (stored in JSON document format). Now Elasticsearch must communicate with the dashboard of Kibana. First, we must connect to Kibana with the Elasticsearch connection parameters, such as node addresses and index names.

Once connected, Kibana uses index patterns to specify the Elasticsearch indices from which the data should be retrieved. These are expressions that allow you to specify one or more Elasticsearch indices. For example, an index pattern called ***air_quality_data-2024-01-05*** will contain all the data on the air quality measurement of January 5th.

Now that Elasticsearch is connected to Kibana, it is possible to create different graphs like bar graphs, maps, tables...

The advantage of Kibana is that the user can interact with the graphs using search queries on the indexed data (search for a specific field, a range of values, specific terms...)

5.8 Apache NiFi to Apache Kafka

Apache NiFi will perform data reading, transformation and writing operations. For Kafka, it will manage the data flow so that it reaches its destination optimally

First, we need to configure the NiFi processors. Three types of processors are essential. The source processors will read the data from the sensors. After reading, the transformation processors will transform the data to put it in a format compatible with Kafka (JSON). The Sink processors will be used to send the transformed data to Kafka.

Finally, NiFi publishes the data on specific Kafka topics and is sent as continuous streams. Kafka stores them in partitions and replicates them on multiple brokers to ensure fault tolerance.

5.9 Web Server Applications to Dashboard

Applications allow data to be visualized on their page and not on Kibana, facilitating a better user experience, more fluid and more adapted to different needs.

They will allow users to consult and interact with data via an online interface. In addition, as the interfaces are connected to Kibana, it will be possible to use all the features to be able to create customizable dashboards for the user.

Web server applications send HTTP requests to the Kibana REST API to retrieve data. These requests specify exactly what should be displayed (dashboard, visualization, filters, etc.)

Kibana receives these requests and retrieves the information using Elasticsearch. Then, it returns a response to the web server in the form of a JSON document, containing all the requested data.

Once the web server application receives the JSON data, it uses it to create dynamic visualizations on the web page. Users can then interact with these dashboards.

VI. Bonus Points: Implementing Architecture Parts

In this section, we will explore concrete examples and technical details of the implementation of different parts of the architecture of our real-time air quality monitoring system. These implementations aim to illustrate how the different components of our architecture interact to collect, process, store and visualize air quality data.

We will cover the following points:

- **Kafka Producer and Consumer:** We will show how sensors send data to Kafka (producer) and how this data is consumed by other components of the system (consumer).
- **HDFS Data Ingestion:** We will detail the process of ingesting data from Kafka into HDFS for long-term storage.
- **Spark Query Example:** We will provide an example of a Spark query to process and analyse real-time air quality data.
- **Integration with HBase:** We will explain how analysis results are stored in HBase for quick access.
- **Orchestration with NiFi:** We will illustrate how Apache NiFi is used to orchestrate data flows between the various components of the system.
- **Searching with Elasticsearch**: We will show how data is indexed in Elasticsearch to enable fast and efficient searches.

These practical examples will provide a better understanding of the interactions between the different technologies used in our architecture and demonstrate the robustness and effectiveness of our air quality monitoring system.

6.1 Kafka Producer and Consumer

In this subsection, we will explore how to implement a Kafka producer that sends air quality data and a Kafka consumer that reads this data. Kafka plays a crucial role in our architecture by enabling reliable, real-time transmission of data between sensors and processing systems.

For the Kafka producer, we use the Kafka Producer library in Java. The producer sends JSON messages containing air quality data to a Kafka topic named ‘air-quality-data’. Each message is associated with a unique key representing the sensor identifier.

```java
import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class KafkaProducerExample {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);

        String topic = "air-quality-data";
        String key = "sensor1";
        String value = "{\"pm25\": 15, \"pm10\": 30, \"o3\": 40, \"timestamp\": 1620000000}";

        producer.send(new ProducerRecord<>(topic, key, value));
        producer.close();
    }
}
```

For the Kafka consumer, we use Java's Kafka Consumer library. The consumer reads messages from the 'air-quality-data' topic and displays them to the console. The consumer is configured to be part of a group of consumers called 'air-quality-group'.

```java
import org.apache.kafka.clients.consumer.*;
import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

public class KafkaConsumerExample {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "air-quality-group");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

        Consumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("air-quality-data"));

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());
            }
        }
    }
}
```

6.2 HDFS Data Ingestion

In this sub-section, we will detail the process of ingesting Kafka data into HDFS for long-term storage. HDFS is used to store large amounts of data in a reliable and scalable way.

To ingest data into HDFS, we use Java's Hadoop FileSystem library. The following code shows how to write Kafka data to an HDFS file named 'raw-data.json'.

```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FSDataOutputStream;

public class HDFSIngestionExample {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://localhost:9000");
        FileSystem fs = FileSystem.get(conf);

        Path hdfsPath = new Path("/air-quality-data/raw-data.json");
        FSDataOutputStream outputStream = fs.create(hdfsPath);

        // Écrire les données de Kafka dans HDFS
        String kafkaMessage = "{\"pm25\": 15, \"pm10\": 30, \"o3\": 40, \"timestamp\": 1620000000}";
        outputStream.writeBytes(kafkaMessage);
        outputStream.close();
    }
}
```

6.3 Spark Query Example

In this subsection, we will provide an example of a Spark query for processing and analyzing air quality data in real time. Spark is used for its real-time and distributed processing capabilities, which are essential for analyzing air quality data.

The following code shows how to read JSON data from HDFS, process it to calculate pollutant averages per sensor, and display the results.

```scala
import org.apache.spark.sql.SparkSession

object AirQualityAnalysis {
    def main(args: Array[String]): Unit = {
        val spark = SparkSession.builder().appName("AirQualityAnalysis").getOrCreate()

        val df = spark.read.json("hdfs://localhost:9000/air-quality-data/")

        val avgPollutants = df.groupBy("sensor_id")
            .agg(
                avg("pm25").alias("avg_pm25"),
                avg("pm10").alias("avg_pm10"),
                avg("o3").alias("avg_o3")
            )

        avgPollutants.show()
    }
}
```

6.4 Integration with HBase

In this sub-section, we will explain how analysis results are stored in HBase for quick access. HBase is a NoSQL database that enables fast data storage and retrieval.

The following code shows how to store analysis results in an HBase table named 'air_quality_results'.

```java
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

public class HBaseIntegrationExample {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionFactory.createConnection();
        Table table = connection.getTable(TableName.valueOf("air_quality_results"));

        Put put = new Put(Bytes.toBytes("sensor1"));
        put.addColumn(Bytes.toBytes("data"), Bytes.toBytes("avg_pm25"), Bytes.toBytes("15.5"));
        put.addColumn(Bytes.toBytes("data"), Bytes.toBytes("avg_pm10"), Bytes.toBytes("30.2"));

        table.put(put);
        table.close();
        connection.close();
    }
}
```

6.5 Orchestration with NiFi

In this subsection, we will illustrate how Apache NiFi is used to orchestrate data flows between the various components of the system. NiFi enables data flows to be managed and transformed flexibly and efficiently.

An example of NiFi data flow orchestration:

1. Create a 'ConsumeKafka' processor to read data from Kafka.
2. Add a 'SplitJson' processor to split JSON messages.
3. Use 'PutHDFS' to store raw data in HDFS.
4. Configure 'ExecuteSparkJob' to launch Spark analysis.
5. Use 'PutHBaseJSON' to store the results in HBase.

6.6 Search with Elasticsearch

In this sub-section, we will show how data is indexed in Elasticsearch to enable fast and efficient searches. Elasticsearch is used for its real-time search and analysis capabilities.

The following code shows how to index data in Elasticsearch and perform a search.

```java
import org.elasticsearch.action.index.IndexRequest;
import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.action.search.SearchRequest;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.RequestOptions;
import org.elasticsearch.client.RestClient;
import org.elasticsearch.common.xcontent.XContentType;
import org.elasticsearch.index.query.QueryBuilders;
import org.elasticsearch.search.builder.SearchSourceBuilder;

import java.io.IOException;

public class ElasticsearchExample {
    public static void main(String[] args) throws IOException {
        RestClient client = RestClient.builder(new HttpHost("localhost", 9200, "http")).build();

        String jsonString = "{\"sensor_id\": \"sensor1\", \"pm25\": 15, \"pm10\": 30, \"o3\": 40, \"timestamp\": 1620000000}";
        IndexRequest request = new IndexRequest("air_quality").id("1").source(jsonString, XContentType.JSON);
        IndexResponse indexResponse = client.index(request, RequestOptions.DEFAULT);

        SearchRequest searchRequest = new SearchRequest("air_quality");
        SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder().query(QueryBuilders.matchQuery("sensor_id", "sensor1"));
        searchRequest.source(searchSourceBuilder);

        SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);
        System.out.println(searchResponse);
    }
}
```

VII. Conclusion

The implementation of a real-time air quality monitoring system is a crucial initiative for responding to the environmental and public health challenges posed by air pollution. This project has demonstrated how an architecture based on cutting-edge technologies such as Apache Kafka, Hadoop HDFS, Apache Spark, HBase, Apache NiFi and Elasticsearch can be used to collect, process, store and visualize air quality data in an efficient and scalable manner.

The implementations detailed in this section show how each component of the architecture interacts to ensure reliable data transmission, real-time processing, robust storage and interactive visualization. The code examples provided for Kafka producers and consumers, data ingestion in HDFS, Spark queries, integration with HBase, orchestration with NiFi and search with Elasticsearch illustrate the feasibility and effectiveness of this architecture.

The system not only enables real-time monitoring of pollution levels in different geographical areas but also forecasts future trends using advanced analyses. The results are presented clearly and interactively via dashboards, making it easier for local authorities to take decisions and raise public awareness of air quality issues.

In conclusion, this project highlights the importance of integrating big data technologies for environmental monitoring. The proposed solutions provide a solid basis for future improvements and can be adapted to other environmental monitoring contexts, thereby contributing to a healthier and more sustainable future.

VIII. Bibliography

9.1 Technical Documentation

The technical documentation used for this project includes the official guides and resources for the technologies employed. These documents were essential for understanding the functionalities, configurations and best practices for each component of the architecture Apache Kafka:

- **Apache Kafka**
  - Documentation officielle d'Apache Kafka
  - Guide de démarrage rapide pour Kafka
- **Hadoop HDFS :**
  - Documentation officielle d'Hadoop HDFS
  - Guide de configuration de HDFS
- **Apache Spark :**
  - Documentation officielle d'Apache Spark
  - Guide de programmation de Spark
- **Apache HBase :**
  - Documentation officielle d'Apache HBase
  - Guide de configuration d'HBase
- **Apache NiFi :**
  - Documentation officielle d'Apache NiFi
  - Guide de l'utilisateur de NiFi
- **Elasticsearch :**
  - Documentation officielle d'Elasticsearch
  - Guide de démarrage rapide pour Elasticsearch

9.2 Articles and Case Studies

The following articles and case studies were consulted to enrich our understanding of the technologies used and to identify best practice in similar contexts.

- **Articles on Air Quality Monitoring:**
  - World Journal of Research and Review

    - World Journal of Research and Review
- **Theses and Academic Studies:**
    - Thèse de Sylvain Poupry sur la Surveillance de la Qualité de l'Air
- **NoSQL database comparisons:**
    - Comparaison entre HBase, Cassandra et MongoDB
- **Monitoring and Observability:**
    - Monitoring d'Apache HBase avec Instana
    - Documentation Google Cloud sur le monitoring d'HBase
    - Monitoring d'Apache HBase avec IBM Instana
- **Case studies and tutorials:**
    - Suivi de la Qualité de l'Air avec HDP et HDF
    - Exemple d'architecture de système de surveillance
    - Exemple de producteur et consommateur Kafka
    - Accéder aux données HDFS avec Cloudera Data Science Workbench

9.3. Videos

The following videos were consulted to enrich our understanding of the technologies used and how to connect them.

- **Apache Kafka:**
    - Getting Started with Apache Kafka
    - Apache Kafka Tutorial for Beginners
- **Hadoop HDFS:**
    - Hadoop HDFS Basics
- **Apache HBase:**
    - Getting Started with Apache HBase
    - Apache HBase Tutorial
- **Apache NiFi:**
    - Apache NiFi Basics
    - Apache NiFi Tutorial