Design and Implementation of a Distributed Analytics Platform Using Apache Cassandra

# Design & Architecture

**Overall System Design**

The distributed analytics platform is architected to handle high-velocity IoT sensor data streams in real-time. It integrates fault tolerance, scalability, and high availability while maintaining low-latency query execution. The architecture consists of three core components:

**Cassandra Cluster**: A multi-node cluster ensuring distributed data storage and fault tolerance through replication. Nodes are distributed across data centers to mitigate regional failures.  
Nodes are distributed across a bridge network, enabling inter-node communication while isolating the cluster from external disruptions.

**Data Ingestion Pipeline**: Apache Kafka is employed for real-time streaming, enabling data ingestion from IoT devices at scale. Python scripts simulate sensor data and publish it to Kafka topics.

**Query Engine**: Optimized for time-series data analytics and ad-hoc queries.   
Query strategies include partitioning, clustering, materialized views, and secondary indexes.  
------------------------------------------------------------------------------------------------------------------------------------------

**Network Topology**

**Bridge Network**: cassandra-sensor-network provides a secure and isolated communication layer among the components.

**Nodes Configuration:**

* **Seed Node:** Responsible for discovery and communication initialization.
* **Data Nodes:** Three additional Cassandra nodes provide replication and data partitioning.
* **Zookeeper Node:** Coordinates Kafka nodes and ensures distributed consistency.
* **Kafka Node:** Handles real-time data ingestion and message streaming.

------------------------------------------------------------------------------------------------------------------------------------------

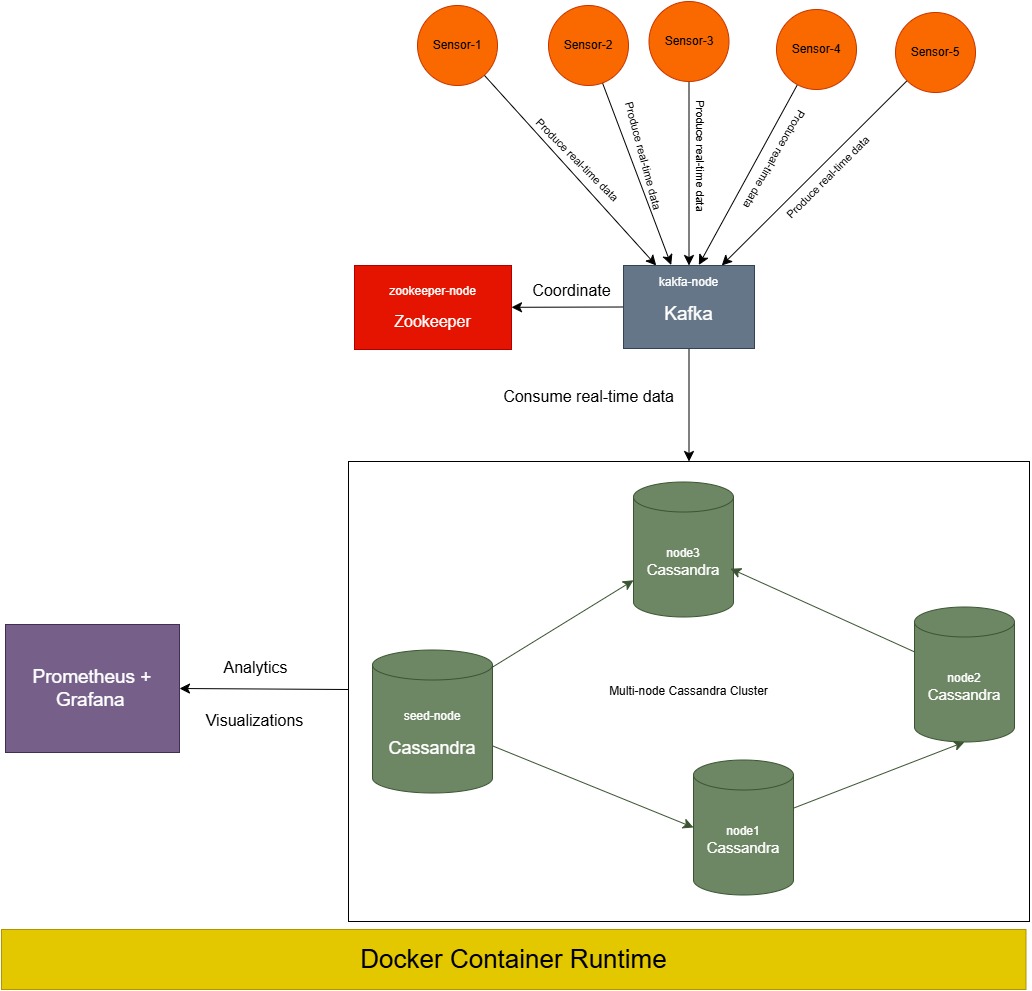
**Fault Tolerance and Scalability**

* **Fault Tolerance**: Cassandra’s built-in replication strategy ensures data is consistently available, even if multiple nodes fail. The gossip protocol dynamically detects and handles such failures.
* **Scalability**: The architecture supports horizontal scaling. Additional nodes can be added dynamically to accommodate increased workloads, with Cassandra redistributing data to maintain balance.

------------------------------------------------------------------------------------------------------------------------------------------

**Role of Cassandra in Architecture**

* **Primary Database**: Apache Cassandra is used for its high write and read throughput, ideal for time-series IoT sensor data.
* **Data Replication**: Configured to replicate data across nodes to ensure durability and prevent data loss.
* **Query Optimization**: Cassandra’s partition and clustering keys enable efficient time-range queries, critical for IoT data analysis.



# Data Modeling

**Data Model for the Use Case**

To handle IoT sensor data effectively, a schema optimized for time-series data was designed. Each record represents a sensor reading, and the schema allows efficient retrieval of data over specific time ranges.

**Table:** sensor\_readings

* sensor\_id (Partition Key): Ensures even data distribution across nodes.
* timestamp (Clustering Key): Orders data within a partition for efficient time-range queries.
* reading\_value: Stores the sensor reading.
* location: Records the sensor’s location.

CREATE TABLE sensor\_readings (

sensor\_id TEXT,

timestamp TIMESTAMP,

reading\_value FLOAT,

location TEXT,

PRIMARY KEY (sensor\_id, timestamp)

) WITH CLUSTERING ORDER BY (timestamp DESC);

**Optimization Through Partition and Clustering Keys**

* **Partition Keys**: sensor\_id ensures that data for each sensor is distributed evenly across nodes, balancing the load and improving scalability.
* **Clustering Keys**: timestamp allows efficient retrieval of time-ordered data, essential for anomaly detection.
* **Materialized Views**: Materialized views pre-compute frequently used queries, such as those based on temperature ranges or locations, improving query performance.

# Implementation

Architecture

* Seed Node: Facilitates discovery and communication between other nodes.
* Data Replication: Configured replication factor of 3 for fault tolerance.
* Networking: All nodes are part of the same Docker bridge network, enabling inter-node communication.

Deployment - docker-compose simplifies containerized deployment and ensures consistent environments across nodes. - defines the cluster with services for seed node and additional nodes.

services:

seed-node:

image: cassandra

container\_name: seed-node

networks:

- cassandra-sensor-network

ports:

- "9042:9042"

environment:

CASSANDRA\_CLUSTER\_NAME: "CassandraCluster"

CASSANDRA\_NUM\_TOKENS: 256

node1:

image: cassandra

container\_name: node1

networks:

- cassandra-sensor-network

environment:

CASSANDRA\_SEEDS: seed-node

CASSANDRA\_CLUSTER\_NAME: "CassandraCluster"

node2:

image: cassandra

container\_name: node2

networks:

- cassandra-sensor-network

environment:

CASSANDRA\_SEEDS: seed-node

CASSANDRA\_CLUSTER\_NAME: "CassandraCluster"

node3:

image: cassandra

container\_name: node3

networks:

- cassandra-sensor-network

environment:

CASSANDRA\_SEEDS: seed-node

CASSANDRA\_CLUSTER\_NAME: "CassandraCluster"

zookeeper-node:

image: bitnami/zookeeper

container\_name: zookeeper-node

networks:

- cassandra-sensor-network

environment:

ALLOW\_ANONYMOUS\_LOGIN: "yes"

kafka-node:

image: bitnami/kafka

container\_name: kafka-node

networks:

- cassandra-sensor-network

ports:

- "9092:9092"

environment:

ALLOW\_PLAINTEXT\_LISTENER: "yes"

KAFKA\_CFG\_LISTENERS: PLAINTEXT://0.0.0.0:9092

KAFKA\_CFG\_ZOOKEEPER\_CONNECT: "zookeeper-node:2181"

KAFKA\_CFG\_ADVERTISED\_LISTENERS: PLAINTEXT://localhost:9092

KAFKA\_CREATE\_TOPICS: "kafka-topic:1:1"

networks:

cassandra-sensor-network:

driver: bridge

**Real-Time Ingestion and Querying Mechanisms**

**Data Ingestion**: A Python-based ingestion script connects to the Cassandra cluster and writes sensor readings in real-time.

**Kafka Producer (producer.py)**

* Simulates real-time IoT sensor data using Python’s random and uuid libraries.
* Publishes JSON-formatted data to the Kafka topic.

# Kafka configuration

KAFKA\_TOPIC = "kafka-topic"

KAFKA\_BROKER = "localhost:9092"  # Kafka broker address

# Kafka producer configuration

producer\_config = {

    "bootstrap.servers": KAFKA\_BROKER  # Kafka broker connection

}

producer = Producer(producer\_config)

# Function to ingest high-velocity sensor data (simulating real-time data)

def simulate\_data():

    return {

        "sensor\_id": random.choice(['sensor-1', 'sensor-2', 'sensor-3', 'sensor-4', 'sensor-5']),

        "timestamp": datetime.utcnow().isoformat(),  # Use current UTC time

        "temperature": uniform(20, 30),  # Random temp between 20°C and 30°C

        "humidity": uniform(40, 60),  # Random humidity between 40% and 60%

    }

def ingest\_data\_report(err, msg):

    if err is not None:

        print(f"Message delivery failed: {err}")

    else:

        print(f"Message delivered to {msg.topic()} [{msg.partition()}]")

if \_\_name\_\_ == "\_\_main\_\_":

    print(f"Producing messages to Kafka topic '{KAFKA\_TOPIC}'...")

    try:

        while True:

            sensor\_data = simulate\_data()

            sensor\_data\_json = json.dumps(sensor\_data)

            producer.produce(

                KAFKA\_TOPIC,

                key=sensor\_data["sensor\_id"],

                value=sensor\_data\_json,

                callback=ingest\_data\_report

            )

            producer.flush()

            print(f"Produced data: {sensor\_data}")

            time.sleep(2)

    except KeyboardInterrupt:

        print("Shutting down producer...")

    finally:

        producer.flush()  # Ensure all messages are sent before exit

**Kafka Consumer (consumer.py)** - Consumes messages from the Kafka topic, parses JSON data, and inserts it into the Cassandra database.

Kafka Configuration

KAFKA\_TOPIC = "kafka-topic"

KAFKA\_BROKER = "localhost:9092"

GROUP\_ID = "consumer-group"

Cassandra configuration

CASSANDRA\_KEYSPACE = "sensor\_data\_keyspace"

CASSANDRA\_TABLE = "sensor\_data"

Cassandra Setup

def cassandra\_setup():

    cluster = Cluster(['127.0.0.1'])  # Replace with your Cassandra IP

    session = cluster.connect()

    # Create keyspace and table if not exist

    session.execute(f"""

        CREATE KEYSPACE IF NOT EXISTS {CASSANDRA\_KEYSPACE}

        WITH replication = {{ 'class': 'SimpleStrategy', 'replication\_factor': 3 }};

    """)

    session.set\_keyspace(CASSANDRA\_KEYSPACE)

    session.execute(f"""

        CREATE TABLE IF NOT EXISTS {CASSANDRA\_TABLE} (

            sensor\_id TEXT,

            timestamp TIMESTAMP,

            temperature FLOAT,

            humidity FLOAT,

            PRIMARY KEY (sensor\_id, timestamp)

        ) WITH CLUSTERING ORDER BY (timestamp DESC);

    """)

    return session

Insertion Logic:

def insert\_data\_to\_cassandra(session, data):

    session.execute(f"""

        INSERT INTO {CASSANDRA\_TABLE} (sensor\_id, timestamp, temperature, humidity)

        VALUES (%s, %s, %s, %s)

    """, (data["sensor\_id"], data["timestamp"], data["temperature"], data["humidity"]))

**Querying**: Real-time querying of sensor data for specific conditions (e.g., time range).

rows = session.execute("""

SELECT \* FROM sensor\_readings WHERE sensor\_id = %s AND timestamp > %s

""", (sensor\_id, '2024-12-01T00:00:00'))

for row in rows:

print(f"Reading: {row.reading\_value} at {row.location}")

**Data Replication and High Availability**

* **Replication**: Configured with a replication factor of 3 to ensure fault tolerance.
* **Consistency**: Utilized QUORUM for critical reads and writes, balancing consistency and performance.

**Consistency and Fault Tolerance**

* Cassandra handles consistency using tunable consistency levels (e.g., ONE, QUORUM, ALL).
* Gossip protocol ensures that even in node failures, the cluster remains operational by redirecting traffic and repairing data inconsistencies.

**Dockerized Deployment**

All components (Kafka, Zookeeper, Cassandra nodes) were containerized using Docker Compose, ensuring consistent deployment and ease of scaling.

Deploying docker containers: docker-compose up -d

**Scalability Tests**

**Adding New Node**

Scaling horizontally involves adding more nodes to the cluster. This process can be managed by updating the docker-compose.yml file and properly configuring the new nodes.

**Update docker-compose.yml:** Add a new service for the new Cassandra node. For example:

node4:

image: assandra

container\_name: node4

networks:

- assandra-sensor-network

ports:

- “7203:7199” # JMX port

- “9046:9042” # Native transport port

environment:

CASSANDRA\_SEEDS: seed-node

CASSANDRA\_CLUSTER\_NAME: "CassandraCluster"

CASSANDRA\_DC: "DataCenter1"

Start the Updated Cluster and verify the new node in the cluster

docker exec -it seed-node nodetool status

**Removing Nodes**

**Decommission the Node:** Use nodetool decommission on the node you want to remove. This process moves the node's data to other nodes in the cluster.

docker exec -it node3 nodetool decommission

**Verify the Decommission:** Check the cluster's status to ensure the node is no longer part of it:

docker exec -it seed-node nodetool status

# Query Performance

**Efficient Query Execution**

* **Time-Series Optimization**:

Partition and clustering keys ensure efficient query execution for time-range and per-sensor queries.

Example Query:

SELECT \* FROM sensor\_data

WHERE sensor\_id = 'sensor-5'

AND timestamp > '2024-12-02T18:42:50'

AND timestamp < '2024-12-02T18:43:00';

A screenshot of a computer screen

Description automatically generated

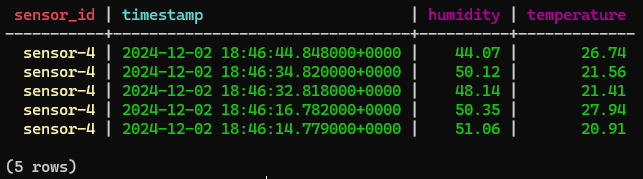
* **Retrieve recent data for a specific sensor**

SELECT \* FROM sensor\_data

WHERE sensor\_id = 'sensor-4'

ORDER BY timestamp DESC

LIMIT 5;



* **Materialized Views**: Pre-computed views enhance query performance for recurring patterns, such as fetching the latest reading by location.

CREATE MATERIALIZED VIEW sensor\_data\_by\_temperature AS

SELECT sensor\_id, timestamp, temperature, humidity

FROM sensor\_data\_keyspace.sensor\_data

WHERE temperature IS NOT NULL

PRIMARY KEY (temperature, timestamp, sensor\_id);

SELECT \* FROM sensor\_data\_by\_temperature

WHERE temperature > 25.0 AND temperature < 30.0;

* **Secondary Indexes**: Indexes are used for queries based on non-primary key attributes.

CREATE INDEX IF NOT EXISTS humidity\_index ON sensor\_data (humidity);

* **Aggregations**: Efficient execution of queries like finding the average reading for a sensor over a time period.

def calculate\_aggregates(session):

try:

query = """ SELECT AVG(temperature) AS avg\_temp, AVG(humidity) AS avg\_hum

FROM sensor\_data; """

rows = session.execute(query)

for row in rows:

print(f"Average Temperature: {row.avg\_temp}, Average Humidity: {row.avg\_hum}")

except Exception as e:

print(f"Error calculating aggregates: {e}")

SELECT AVG(temperature) AS avg\_temp, AVG(humidity) AS avg\_hum FROM sensor\_data;



SELECT MAX(temperature) AS max\_temperature

FROM sensor\_data

WHERE sensor\_id = 'sensor-1';



SELECT sensor\_id, COUNT(\*) AS total\_readings

FROM sensor\_data\_keyspace.sensor\_data

GROUP BY sensor\_id;

A screen shot of a computer

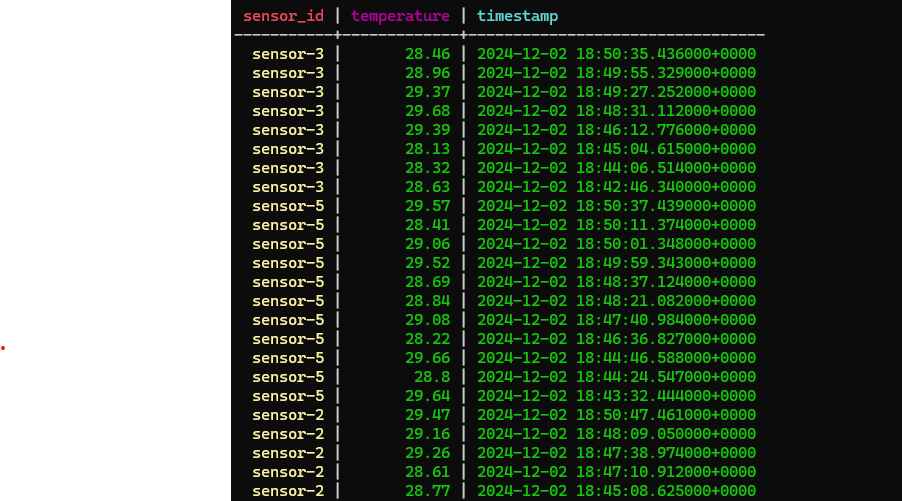
Description automatically generated

* **Anomaly Detection**: Queries for detecting outliers or abnormal readings using predefined thresholds.

SELECT sensor\_id, temperature, timestamp

FROM sensor\_data

WHERE temperature > 28.0;



* **Set TTL**: Expire old data automatically to manage storage efficiently

def set\_ttl(session):

try:

query = """

INSERT INTO sensor\_data (sensor\_id, timestamp, temperature, humidity)

VALUES (%s, %s, %s, %s)

USING TTL 86400; -- 1 day TTL

"""

sensor\_id = random.choice(['sensor-1', 'sensor-2', 'sensor-3', 'sensor-4', 'sensor-5']),

timestamp = datetime.utcnow()

temperature = random.uniform(20.0, 35.0)

humidity = random.uniform(30.0, 70.0)

session.execute(query, (sensor\_id, timestamp, temperature, humidity))

print(f"Inserted data with TTL: Sensor ID={sensor\_id}, Temp={temperature}, Humidity={humidity}")

except Exception as e:

print(f"Error setting TTL: {e}")

INSERT INTO sensor\_data\_keyspace.sensor\_data (sensor\_id, timestamp, temperature, humidity)

VALUES ('sensor-2', '2024-12-02T18:43:00', 25.0, 50.0)

USING TTL 86400;  -- 1 day

**Performance Metrics**

* **Monitoring Tools**

nodetool tablestats sensor\_data\_keyspace

A screenshot of a computer

Description automatically generated

# Monitoring with prometheus and grafana

**Prometheus**: Collects real-time metrics such as read/write latencies.

* Prometheus scrapes metrics from each Cassandra node via the JMX exporter.
* Persistent volumes ensure data retention and prevent loss on container restarts.

Access: <http://localhost:9090>

**Grafana**: Visualizes system health and performance, enabling proactive alerting.

* Links to Prometheus as a data source for visualizing metrics.
* Allows creating dashboards to monitor Cassandra cluster health, latency, and performance metrics

Access: <http://localhost:3000>

prometheus-node:

    image: prom/prometheus

    container\_name: prometheus

    volumes:

      - ./prometheus.yml:/etc/prometheus/prometheus.yml

      - prometheus\_data:/prometheus-node

    networks:

      - cassandra-sensor-network

    ports:

      - "9090:9090"

grafana-node:

    image: grafana/grafana

    container\_name: grafana

    ports:

      - "3000:3000"

    networks:

      - cassandra-sensor-network

    depends\_on:

      - prometheus-node

    environment:

      - GF\_SECURITY\_ADMIN\_USER=admin

      - GF\_SECURITY\_ADMIN\_PASSWORD=admin

prometheus.yml

global:

  scrape\_interval: 15s

scrape\_configs:

  - job\_name: 'cassandra'

    static\_configs:

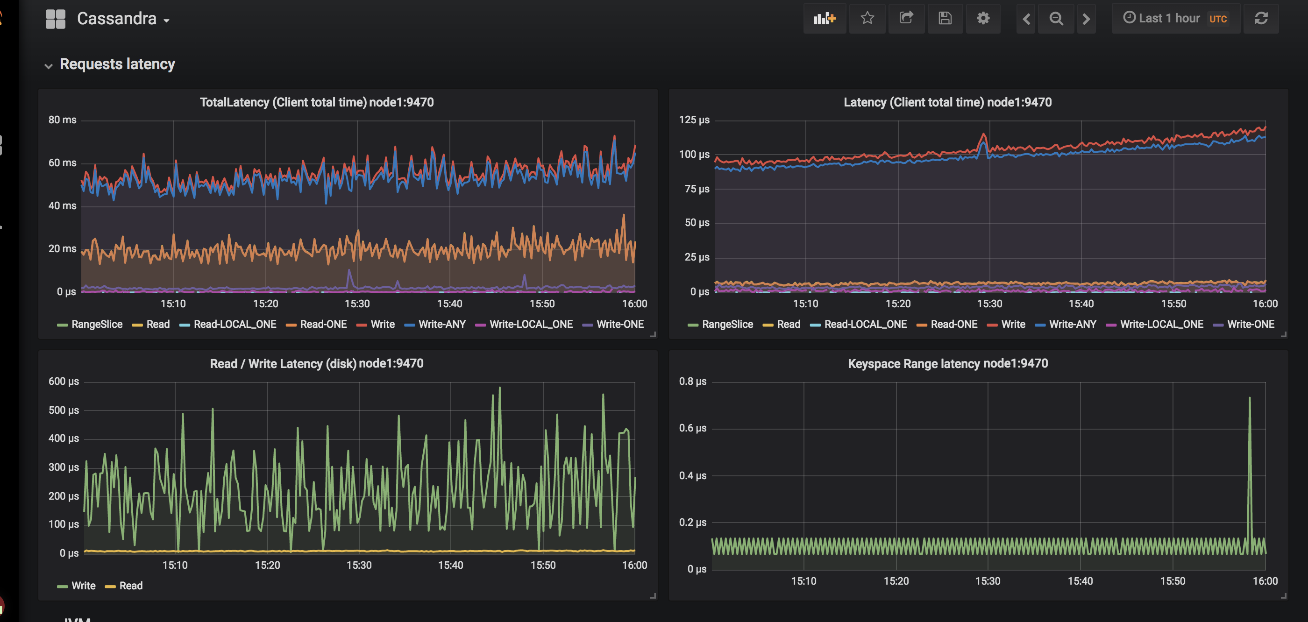
      - targets: ['seed-node:7199', 'node1:7199', 'node2:7199', 'node3:7199']

A black rectangular object with white lines

Description automatically generated

A screenshot of a computer

Description automatically generated





A screenshot of a computer

Description automatically generated

# Summary

This project demonstrates a robust distributed analytics platform capable of handling high-throughput IoT data. By leveraging Apache Cassandra’s distributed architecture and Apache Kafka’s real-time streaming capabilities, the platform achieves fault tolerance, scalability, and efficient query execution. Future enhancements, including advanced analytics and monitoring integration, will further solidify its utility for time-series data analysis.