DESIGN AND IMPLEMENTATION OF A DISTRIBUTED ANALYTICS PLATFORM USING APACHE CASSANDRA

1. 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:

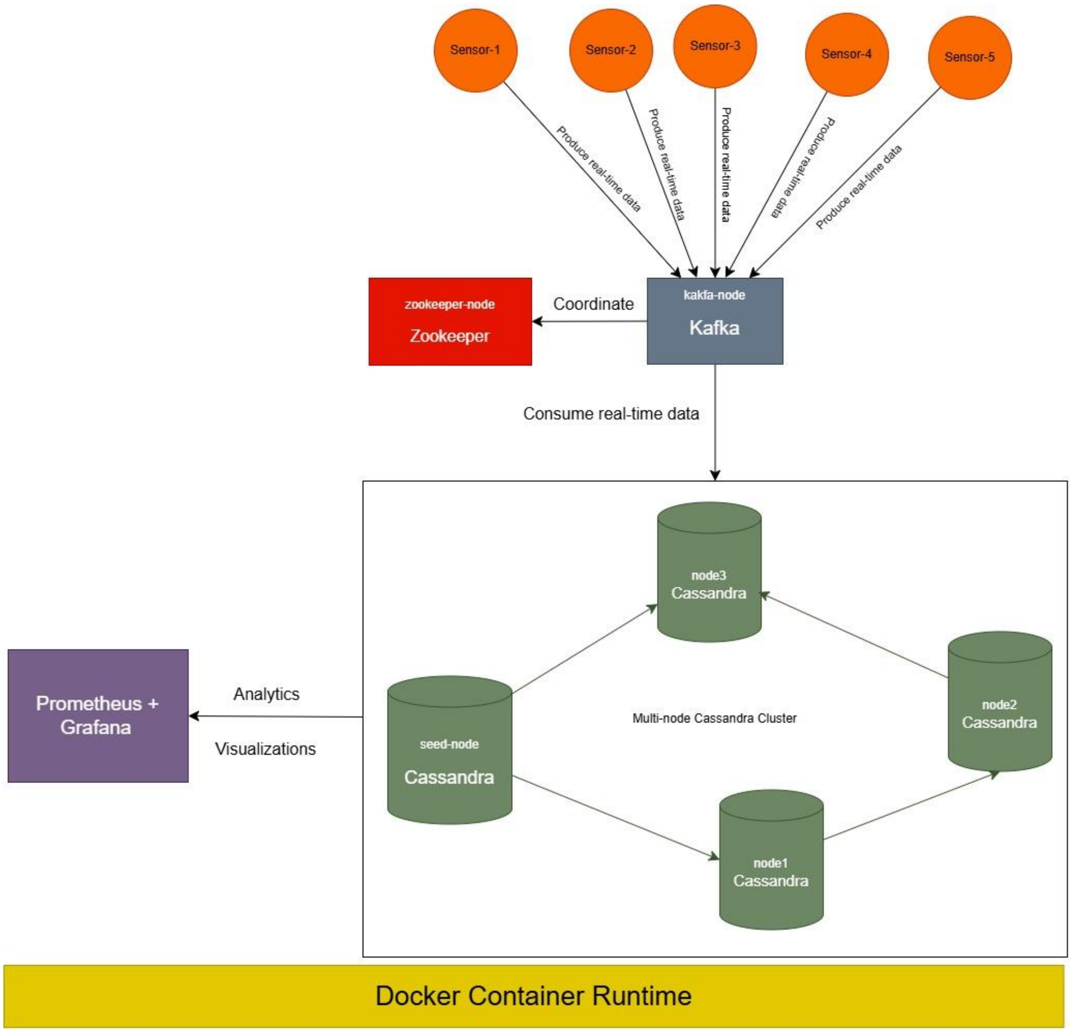
- **Seed Node:** Responsible for discovery and communication initialization.
- **Data Nodes:** Three additional Cassandra nodes provide replication and data partitioning.
- **Zookeeper Node:** Coordinates Kafka nodes and ensures distributed consistency.
- **Kafka Node:** Handles real-time data ingestion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



2. 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.

3. 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
```

4. 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';
```

```
 sensor_id | timestamp                       | humidity | temperature
-----------+---------------------------------+----------+-------------
  sensor-5 | 2024-12-02 18:42:58.362000+0000 |    49.21 |       21.03
  sensor-5 | 2024-12-02 18:42:56.358000+0000 |    59.34 |       23.74
  sensor-5 | 2024-12-02 18:42:54.354000+0000 |    41.05 |       23.16

(3 rows)
```

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

```
SELECT * FROM sensor_data
WHERE sensor_id = 'sensor-4'
ORDER BY timestamp DESC
LIMIT 5;
```

```
 sensor_id | timestamp                       | humidity | temperature
-----------+---------------------------------+----------+-------------
  sensor-4 | 2024-12-02 18:46:44.848000+0000 |    44.07 |       26.74
  sensor-4 | 2024-12-02 18:46:34.820000+0000 |    50.12 |       21.56
  sensor-4 | 2024-12-02 18:46:32.818000+0000 |    48.14 |       21.41
  sensor-4 | 2024-12-02 18:46:16.782000+0000 |    50.35 |       27.94
  sensor-4 | 2024-12-02 18:46:14.779000+0000 |    51.06 |       20.91

(5 rows)
```

- **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;
```

```
 avg_temp | avg_hum
----------+----------
 24.69454 | 49.98188

(1 rows)
```

```
SELECT MAX(temperature) AS max_temperature
FROM sensor_data
WHERE sensor_id = 'sensor-1';
```

```
 max_temperature
-----------------
           29.97

(1 rows)
```

```
SELECT sensor_id, COUNT(*) AS total_readings
FROM sensor_data_keyspace.sensor_data
GROUP BY sensor_id;
```

```
 sensor_id | total_readings
-----------+----------------
  sensor-3 |            103
  sensor-5 |            125
  sensor-2 |             95
  sensor-1 |            126
  sensor-4 |            115

(5 rows)
```

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

```
SELECT sensor_id, temperature, timestamp
FROM sensor_data
WHERE temperature > 28.0;
```

sensor_id	temperature	timestamp
sensor-3	28.46	2024-12-02 18:50:35.436000+0000
sensor-3	28.96	2024-12-02 18:49:55.329000+0000
sensor-3	29.37	2024-12-02 18:49:27.252000+0000
sensor-3	29.68	2024-12-02 18:48:31.112000+0000
sensor-3	29.39	2024-12-02 18:46:12.776000+0000
sensor-3	28.13	2024-12-02 18:45:04.615000+0000
sensor-3	28.32	2024-12-02 18:44:06.514000+0000
sensor-3	28.63	2024-12-02 18:42:46.340000+0000
sensor-5	29.57	2024-12-02 18:50:37.439000+0000
sensor-5	28.41	2024-12-02 18:50:11.374000+0000
sensor-5	29.06	2024-12-02 18:50:01.348000+0000
sensor-5	29.52	2024-12-02 18:49:59.343000+0000
sensor-5	28.69	2024-12-02 18:48:37.124000+0000
sensor-5	28.84	2024-12-02 18:48:21.082000+0000
sensor-5	29.08	2024-12-02 18:47:40.984000+0000
sensor-5	28.22	2024-12-02 18:46:36.827000+0000
sensor-5	29.66	2024-12-02 18:44:46.588000+0000
sensor-5	28.8	2024-12-02 18:44:24.547000+0000
sensor-5	29.64	2024-12-02 18:43:32.444000+0000
sensor-2	29.47	2024-12-02 18:50:47.461000+0000
sensor-2	29.16	2024-12-02 18:48:09.050000+0000
sensor-2	29.26	2024-12-02 18:47:38.974000+0000
sensor-2	28.61	2024-12-02 18:47:10.912000+0000
sensor-2	28.77	2024-12-02 18:45:08.625000+0000

- **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

```
E:\apache-project>docker exec -it node1 bash
root@e6becb87eb16:/# nodetool tablestats sensor_data_keyspace
Total number of tables: 2
----------------
Keyspace: sensor_data_keyspace
        Read Count: 41
        Read Latency: 0.7341219512195122 ms
        Write Count: 1000
        Write Latency: 0.12143300000000001 ms
        Pending Flushes: 0
                Table (index): sensor_data.humidity_index
                SSTable count: 1
                Old SSTable count: 0
                Max SSTable size: 13.580KiB
                Space used (live): 13906
                Space used (total): 13906
                Space used by snapshots (total): 0
                Off heap memory used (total): 384
                SSTable Compression Ratio: 0.41957
                Number of partitions (estimate): 463
                Memtable cell count: 223
                Memtable data size: 9899
                Memtable off heap memory used: 0
                Memtable switch count: 1
                Speculative retries: 0
                Local read count: 0
                Local read latency: NaN ms
                Local write count: 500
                Local write latency: 0.066 ms
                Local read/write ratio: 0.00000
                Pending flushes: 0
                Percent repaired: 0.0
                Bytes repaired: 0B
                Bytes unrepaired: 12.343KiB
                Bytes pending repair: 0B
                Bloom filter false positives: 0
                Bloom filter false ratio: 0.00000
                Bloom filter space used: 336
```

5. 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

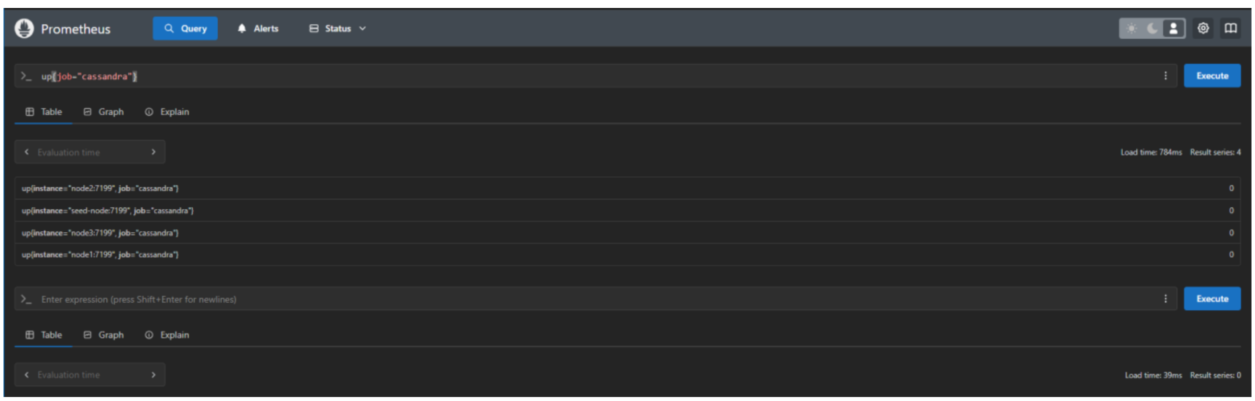
Access: http://localhost:3000

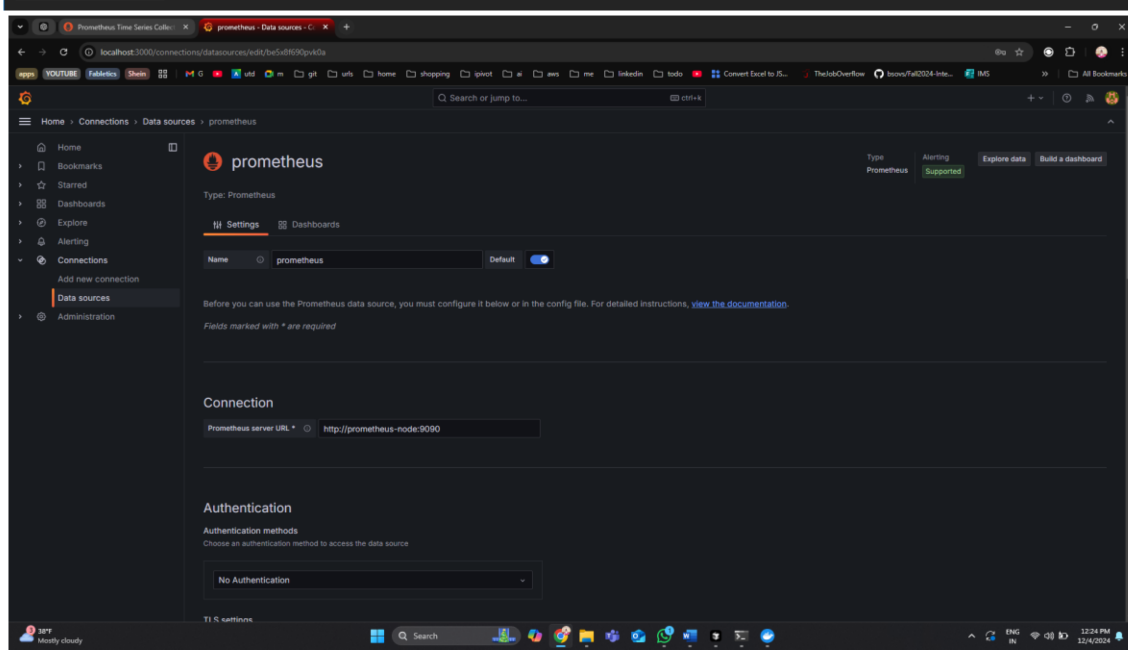
```
  prometheus-node:
    image: prom/prometheus
    container_name: prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus-node
    networks:
      - cassandra-sensor-network
    ports:
```

```yaml
      - "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

```yaml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'cassandra'
    static_configs:
      - targets: ['seed-node:7199', 'node1:7199', 'node2:7199', 'node3:7199']
```






Cassandra
Last 1 hour UTC
Requests latency
TotalLatency (Client total time) node1:9470
Latency (Client total time) node1:9470
RangeSlice Read Read-LOCAL_ONE Read-ONE Write Write-ANY Write-LOCAL_ONE Write-ONE
Read / Write Latency (disk) node1:9470
Write Read
Keyspace Range latency node1:9470


Cassandra
Last 3 hours UTC
instance node1:9470
interval 30s
Node status
UP
ClientWrite avg duration
1.113 ms
ClientWrite avg duration
1.113 ms
JVM Heap usage
10%
Pending compactions
1
Pending threads
1
ParNew GC duration
99.3 ms
OldGen GC duration
90.0 ms
Unavailable exceptions
0
Timeout exceptions
0
Connection timeouts
0
Coordinator level stats (data can be R/W using other nodes)
Writes / sec cassandra-node1:9470
Write Write-ANY Write-LOCAL_ONE Write-ONE
Reads / sec cassandra-node1:9470
Read Read-LOCAL_ONE Read-ONE
Avg write latency cassandra-node1:9470
Avg read latency cassandra-node1:9470


Cassandra
Last 1 hour UTC
GC time
ConcurrentMarkSweep ParNew
Heap usage
Used heap size Max heap size
Memory Pool used
CMS Old Gen Code Cache Compressed Class Space Metaspace
Par Eden Space Par Survivor Space
Requests count/errors (7 hidden panels)
Disk usage (2 hidden panels)
Compaction (6 hidden panels)
Connected clients (1 hidden panels)
Hinted handoff (2 hidden panels)
Bloom filter (4 hidden panels)
Thread pools (6 hidden panels)
Key cache (5 hidden panels)
Memtables (4 hidden panels)
Snapshots (1 hidden panels)

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.