Anomaly Detection Report

Building Management System - Equipment Failure Analysis

1. Introduction

Anomaly detection plays a crucial role in predictive maintenance for building management systems. This report focuses on detecting unusual equipment failures by analysing maintenance logs. The primary goal is to identify anomalies in the resolution time of maintenance issues, which could indicate potential faults, inefficiencies, or abnormal conditions in the system.

2. Anomaly Detection Method Used

The anomaly detection model implemented in this project is based on Isolation Forest (IF), a machine learning algorithm specifically designed for outlier detection.

Why Isolation Forest?

- **Efficiency:** Unlike traditional density-based methods, Isolation Forest is computationally efficient, making it suitable for real-time anomaly detection.
- **Interpretability:** It isolates anomalies instead of profiling normal behaviour, making it easier to detect rare but significant events.
- **Scalability:** The model scales well to large datasets, making it ideal for continuous monitoring of equipment failures.

How Isolation Forest Works?

Isolation Forest detects anomalies by randomly partitioning data points using decision trees. Anomalies are isolated faster compared to normal data points, as they require fewer splits in the tree.

The model assigns anomaly scores based on how early a point gets isolated. Points with the highest anomaly scores are flagged as outliers.

3. Main Code for Anomaly Detection

The following code implements Isolation Forest to detect anomalies in maintenance logs. The input data contains equipment failure events, including the resolution time for each issue. If a maintenance event takes an unusually long time to resolve, it is flagged as an anomaly.

```
from sklearn.ensemble import IsolationForest

import pandas as pd

def detect_anomalies(df):

    """Detects anomalies in maintenance resolution time using Isolation Forest"""

    model = IsolationForest(contamination=0.05, random_state=42)

    df["anomaly"] = model.fit_predict(df[["resolution_time_hours"]])
```

```
    # Flag anomalies (-1 indicates an anomaly)
    anomalies = df[df["anomaly"] == -1]
    return anomalies
# Sample maintenance logs
data = {
    "event_id": [f"E{i}" for i in range(100)],
    "equipment": ["HVAC"] * 50 + ["Electrical"] * 50,
    "issue": ["Overheating"] * 30 + ["Power Failure"] * 70,
    "resolution_time_hours": [2 + (i % 5) * 10 for i in range(100)]
}
# Convert to DataFrame
df = pd.DataFrame(data)
# Detect anomalies
anomalies = detect_anomalies(df)
print("Detected Anomalies:\n", anomalies)
```

4. Conclusion

The Isolation Forest algorithm effectively identifies abnormal equipment failures by analysing resolution times. Anomalies detected by this model indicate potential maintenance inefficiencies, faults, or unusual repair delays. Implementing this method in a real-time monitoring system can improve predictive maintenance and reduce equipment downtime in building management systems.

Query Optimization Strategy for Building Management System

1. Introduction

Query optimization is a crucial aspect of database management, ensuring that data retrieval is **efficient, scalable, and cost-effective**. In a **building management system (BMS)**, where multiple databases (MongoDB, SQL, Time-Series, and Vector DB) interact, an optimized query execution strategy is essential for:

- **Reducing latency** in retrieving equipment failure logs.
- **Minimizing computational overhead** for real-time anomaly detection.
- **Ensuring smooth integration** between the **query parser, executor, and databases**.

This document outlines an **effective query optimization strategy** for handling structured queries generated by an **NLP query parser** and optimizing execution across different database types.

2. Understanding the Query Flow in the System

The system architecture consists of the following components:

1. **NLP Query Parser (Python + OpenAI API):** Converts user queries into structured database queries.
2. **Query Executor (Python):** Processes structured queries and determines the appropriate database.
3. **Multiple Databases:**
    - **MongoDB (Cosmos DB):** Stores logs and maintenance records.
    - **SQL Database:** Maintains financial and report-related data.
    - **Time-Series Database:** Handles sensor data such as temperature, power consumption, and HVAC performance.
    - **Vector Database:** Stores document-based reports for retrieval and analysis.

3. Query Optimization Techniques

3.1. Indexing for Faster Query Execution

Indexing improves **search efficiency** by reducing the time required to fetch data from large datasets.

- **MongoDB:** Create **compound indexes** on frequently queried fields like equipment and timestamp.
- **SQL Database:** Use **B-Trees and Hash Indexes** for structured data storage.
- **Time-Series Database:** Implement **time-based partitioning** to improve retrieval speed.
- **Vector Database:** Use **approximate nearest neighbors (ANN)** for fast similarity searches.

3.2. Query Caching for Reducing Redundant Computation

To avoid repeated execution of the same query, **caching mechanisms** can be implemented:

- **Redis Cache:** Store frequently queried data (e.g., past equipment failures) to reduce database load.
- **Query Result Caching:** Store structured query results for a fixed duration to prevent repetitive computations.
- **Materialized Views in SQL:** Precompute frequently used aggregations (e.g., average repair time per equipment type).

3.3. Query Optimization Techniques Per Database Type

MongoDB (Logs & Maintenance)

- Use **aggregation pipelines** to **filter, group, and transform** maintenance logs efficiently.
- **Sharding strategy:** Distribute data across multiple nodes for faster retrieval.
- Implement **TTL Indexes** to automatically delete old logs beyond a retention period.

SQL Database (Financials & Reports)

- Use **EXPLAIN ANALYZE** to check query execution plans.
- Optimize **JOIN operations** by using indexed columns.
- Apply **denormalization** where necessary to reduce complex multi-table queries.

Time-Series Database (Sensor Data)

- Store data in **optimized time-based partitions** (e.g., hourly, daily).
- Use **downsampling techniques** to reduce data volume for older records.
- Implement **window functions** for fast time-based aggregations.

Vector Database (Document Search)

- Use **Approximate Nearest Neighbour's (ANN) indexing** for efficient document retrieval.
- Optimize **embedding similarity search** by reducing vector dimensionality.

4. Query Execution Plan for Different Scenarios

Scenario 1: Retrieve HVAC Failures in the Last 3 Months

- **Optimized Approach:**
    - Use an **index scan** on equipment and timestamp fields in MongoDB.
    - Apply **aggregation pipeline** to filter records within the date range.

Scenario 2: Analyse Energy Consumption Trends Over the Last 6 Months

- **Optimized Approach:**
    - Query the **Time-Series Database** with a **time-partitioned scan**.
    - Use **rolling aggregations** to reduce computational load.
    - Implement **caching for recent queries** to improve response time.

Scenario 3: Fetch Equipment Maintenance Costs for Q1 Reports

- **Optimized Approach:**
    - Use **indexed joins** in the SQL database to fetch financial data.
    - Precompute and store **aggregated maintenance costs** using materialized views.
    - Apply **query caching** for frequently accessed reports.

5. Conclusion

A **well-optimized query strategy** significantly improves the performance of the building management system by:

- **Reducing query execution time** through indexing and caching.
- **Minimizing database load** with optimized storage techniques.
- **Ensuring fast and scalable retrieval** of equipment logs, financial data, and sensor readings.

By implementing these optimization techniques, the system can handle **real-time anomaly detection** efficiently, ensuring **timely maintenance interventions** and **improving overall building management operations**.