Data Pipeline Documentation

Overview

This document provides a comprehensive guide to the sensor data pipeline, designed to process CSV files in real-time, validate and store data, and support scalability for production use. It meets internship requirements by integrating robust architecture, clear setup instructions, and a scalability strategy.

Architecture and Design

This section details the comprehensive and scalable architecture of the sensor data pipeline, engineered to process files efficiently while ensuring reliability and adaptability for production environments.

Pipeline Components:

Core Implementation: Developed in Python with `pipeline_automated.py`, the pipeline operates as a real-time system. It utilizes the `watchdog` library to continuously monitor the `data` folder, initiating processing for new or existing CSV files upon detection or startup, enabling seamless 24/7 operation.

Data Processing Workflow: The pipeline performs rigorous data validation, including checks for null values and temperature ranges (-50°C to 50°C), transforms data into a standardized format, and inserts raw records into a partitioned database table. It also calculates aggregated metrics (e.g., minimum, maximum, mean, standard deviation) for each sensor type per device, enhancing analytical capabilities.

Fault Tolerance Mechanisms: Integrates `tenacity` for a robust retry mechanism, executing database operations up to 3 times with exponential backoff (4, 7, 10 seconds) to recover from transient failures such as database outages. Errors are meticulously logged to `logs/pipeline_automated.log`, while corrupted or failed files are quarantined in a `failed` folder, preventing pipeline crashes and facilitating post-mortem analysis.

Performance Optimization: Leverages Python's parallel processing capabilities to manage multiple files concurrently, providing a foundation for future distributed scaling and improving throughput.

##Database Design:

Schema Structure: Hosted on PostgreSQL within the `sensor_data` database under the `sensor_schema` namespace, the schema comprises three interconnected tables:

raw_sensor_data_partitioned: A partitioned table storing raw sensor data, with `ts` (timestamp with timezone) as the partition key, optimized for time-series queries. Partitions are configured for July 2020 data in UTC (e.g., `2020-07-01 00:00:00+00` to `2020-08-01 00:00:00+00`) with a `DEFAULT` partition to handle outliers.

aggregated_metrics: Stores precomputed statistical summaries (e.g., min, max, average, standard deviation) per device and sensor type, indexed for rapid access and analysis.

files: A normalization table that maps file metadata to raw data, ensuring data integrity with a unique constraint on file_name.

Constraints and Indexing: Enforces primary keys and unique constraints to eliminate duplicates, complemented by indexes on `ts` and `file_id` to boost query performance.

Storage Efficiency: Partitioning minimizes query overhead, while the design supports future scalability with additional partitions (e.g., monthly or yearly), reducing storage costs and improving maintenance.

Scalability Foundation:

The architecture is inherently designed for horizontal scaling. The continuous monitoring and parallel processing capabilities can seamlessly integrate with distributed systems like Apache Kafka for ingestion and Apache Spark for processing, as detailed in the scalability discussion. The partitioned database layout is well-suited for cloud with auto-scaling, ensuring it can handle increased loads.

Fault Tolerance and Reliability:

Beyond retries and logging, the pipeline employs transaction management (`conn.commit()`/`rollback()`) to guarantee data consistency across operations. The quarantine process isolates problematic files, allowing the pipeline to continue uninterrupted, while the default partition accommodates unexpected data ranges, enhancing overall resilience.

Instructions for Setting Up and Running the Pipeline Locally

Dependencies

Python- Version 3.10 or higher.

Libraries:
- `pandas` for data manipulation.
- `psycopg2-binary` for PostgreSQL connectivity.

- `watchdog` for real-time file monitoring.

- `tenacity` for retry logic.

Database: PostgreSQL (local instance).

Scalability Strategy

To handle millions of files per day (e.g., 1 million files ≈ 38.475 billion rows, 10 million files ≈ 384.75 billion rows), the pipeline can scale horizontally with the following approach:

- **Ingestion Layer**: Implement Apache Kafka to manage high-throughput ingestion, queuing file events from a cloud storage solution like AWS S3 with 100+ partitions to distribute the load across multiple consumers.
- **Processing Layer**: Deploy Apache Spark with 50-500 executors to process data in parallel, handling 38.475 million to 384.75 million rows per day per node, leveraging its distributed computing power for validation and aggregation.
- **Storage Layer**: Enhance the PostgreSQL database on AWS RDS with yearly partitions, data compression, and sharding to manage up to 100 terabytes, ensuring efficient storage and retrieval.
- **Orchestration**: Use Apache Airflow to schedule jobs, monitor performance, and trigger auto-scaling based on workload.

Optimization Techniques

- **Batching**: Process files in batches of 10,000, targeting a latency of under 10 minutes per batch to optimize resource use.
- **Parallelism**: Utilize 1,000 Spark partitions to maximize concurrency, scaling with cluster size.
- **Data Compression**: Apply gzip compression to CSV files before ingestion and use PostgreSQL's TOAST feature to reduce storage to 50-100 terabytes for 384.75 billion rows.
- **Efficient Inserts**: Implement asynchronous batch inserts with connection pooling to handle thousands of rows per operation.
- **Load Balancing**: Deploy read replicas and database sharding to distribute query loads.

Recommended Technologies

- **Alternatives**: For a serverless approach, consider AWS Lambda triggered by S3 events, paired with Google Cloud Pub/Sub for messaging, offering auto-scaling without server management.
- **Monitoring**: Integrate Prometheus and Grafana to track CPU, memory, and I/O, with auto-scaling rules at 80% utilization.
- **Fault Tolerance**: Enhance with Kafka replication and Spark checkpointing to ensure data integrity during failures.

Performance Goals

- **Throughput**: Achieve 4.4 million rows per second for 10 million files per day, scalable to 44 million rows per second for 100 million files per day with additional nodes.
- **Latency**: Maintain under 10 minutes per 10,000-file batch.
- **Scalability**: Support up to 100 million files per day (approximately 3.8475 trillion rows) with a multi-node cluster.