Task 1: Choose a suitable database and store the data in batches

Abstract

The project aimed to create a scalable, maintainable data system for environmental IoT sensor data collected throughout a municipality. Sensors measure parameters like temperature, humidity, smoke, CO, light, and motion. City planners use this data to monitor air quality and respond to thresholds. The goal was to build a prototype capable of handling large data batches, ensuring quality, portability, and future cloud expandability. The system needed to reduce local dependencies via containerization, support reliable, idempotent data ingestion, and include a simple interface for visualization and verification.

Concept and design rationale

During the conception phase, various storage options were compared: relational databases like MySQL and PostgreSQL, time-series systems such as InfluxDB, and NoSQL solutions like MongoDB. Since the sensor structure may evolve, a schema-flexible database was essential. MongoDB was chosen for its document-oriented model, scalability, and native indexing and replication support. Its JSON-like structure suits IoT data that differs across devices.

The dataset used was an open Kaggle IoT sensor dataset. Each record includes a device identifier, timestamp, and several sensor readings. To ensure portability and reproducibility, the entire setup, MongoDB, Mongo Express, and the Python ETL scripts, was containerized using **Docker Compose**. This decision allows the prototype to run identically on any machine or cloud instance without local installation conflicts. Future migration to MongoDB Atlas or Kubernetes clusters would therefore be seamless.

Technical implementation

The implemented system follows a modular pipeline consisting of four components:

1. **Data cleaning (clean_csv.py)**
   The raw CSV dataset is read using Pandas, extra unnamed columns are removed, and headers are normalized to lowercase. Empty rows are dropped, and the cleaned result is written as *cleaned_IoT_data.csv*. This step ensures that downstream scripts receive consistent input and that no malformed data disrupts ingestion.
2. **Batch ingestion (ingest.py)**
   The cleaned file is read in configurable chunks of 50,000 rows to optimize memory use.

Each record becomes a MongoDB document with typed fields. Timestamps are converted to UTC datetimes. A SHA-1 hash of (device + timestamp) is created as the _id to prevent duplicates during re-ingestion.
The script retries transient errors and logs activity via Python's logging, streaming to console and /logs/ingestion.log. A JSON metrics file summarizes the total processed, inserted, skipped, and runtime for monitoring.

3. **Indexing (01-create-indexes.js)**
   On container start, MongoDB automatically executes this initialization script, creating indexes on timestamp, device, and their combination. Indexing significantly improves query performance for analytical or reporting workloads.
4. **Containerization (Dockerfile, docker-compose.yml)**
   The Dockerfile builds a minimal Python 3.11 image containing all dependencies listed in requirements.txt. Docker Compose orchestrates three services:
   - **MongoDB**: primary database with persistent volume storage.
   - **Mongo Express**: web-based interface for data validation.
   - **App**: the Python ETL container running the cleaning and ingestion scripts automatically.
     Environment variables define connection parameters and can be overridden for different deployments. Running docker-compose up --build initializes the whole system and populates the database automatically.

The pipeline is visualized in the accompanying **ETL pipeline diagram**, showing the flow from data cleaning and transformation to storage and indexing inside the Dockerized environment.

Personal reflection

Working on this project provided valuable hands-on experience with the complete data-engineering lifecycle, from conceptual design to implementation and deployment. I gained practical insight into containerization and the interaction between Python applications and database services running in separate Docker containers.

The most significant challenge was ensuring reliable communication between containers during early testing. Initially, connection failures occurred because of misconfigured environment variables and authentication issues. By inspecting container logs and consulting Docker

documentation, I learned to define explicit network links and service names in the docker-compose.yml file, which resolved the issue.

From a personal standpoint, this project reinforced the importance of modular thinking and incremental testing. Breaking down the system into small, testable components made debugging far easier. I also realized how valuable proper documentation is: adding clear comments and expanding the README with input/output examples helped solidify my own understanding and ensured reproducibility for others.

For future projects, especially those involving streaming or real-time data, I plan to apply the same systematic approach: starting with a reliable batch system, adding monitoring early, and gradually scaling to distributed architectures such as Kafka or Spark Streaming.

Conclusion

The completed batch-ingestion system fulfills the requirements of Task 1 by providing a fully functional, portable, and well-documented ETL pipeline. The solution reliably cleans, transforms, and stores IoT sensor data while creating indexes for efficient querying. Through Dockerization, the setup can be reproduced on any platform and extended toward cloud or streaming environments.

This project not only enhanced my technical competence in MongoDB, Docker, and Python-based ETL development but also strengthened my problem-solving and documentation skills. The experience demonstrated the value of structured experimentation, clear modular design, and reproducible workflows, principles that will guide my approach to future data-engineering challenges.

GitHub repository

https://github.com/Dost-DS/task1_mongodb_batch_ingest.git