SuperCourier ETL: Technical Documentation

Made by Angel Gaspard-Fauvelle, LaPoste Data Engineers Teacher Assistant

1. Introduction

This document provides a detailed technical overview of the SuperCourier ETL pipeline. It is intended for developers and data engineers who wish to understand the internal architecture, performance optimization strategies, and development methodologies employed in this project. The primary design goals were **scalability**, **maintainability**, and **reproducibility**, which were achieved through a combination of specific algorithms, architectural patterns, and modern development tools.

2. Architectural Design and Modularity

The project's architecture is intentionally decoupled, separating distinct functionalities into specialized modules. This modular design is the foundation for a clean and maintainable codebase.

- `config.py`: Centralizes all static configurations, such as file paths and data generation defaults. This makes it easy to adjust parameters without searching through the entire codebase.

- `domain.py`: Isolates all business logic, including coefficients and the core delay calculation formula. This separation ensures that if business rules change, the modifications are confined to a single file.
- `data_generators.py`: Contains all logic for creating synthetic source data. This module can be independently run or modified to simulate different data scenarios.
- `etl_pipeline.py`: Orchestrates the core Extract, Transform, and Load steps. It acts as the conductor, calling other modules to perform specific tasks in the correct sequence.
- `file_manager.py`: Handles all file system interactions, such as archiving and deleting files. This abstracts away I/O complexity from the main pipeline logic.
- `main.py`: Serves as the user-facing entry point, managing interactive prompts and high-level orchestration.

This separation of concerns makes the system easier to debug, profile, and extend.

3. Performance Optimization Strategies

To ensure the pipeline can process millions of records efficiently, several key performance optimization techniques were implemented.

3.1. Vectorization with Pandas and NumPy

The most critical optimization is the strict avoidance of iterative loops over DataFrame rows. Instead, all data manipulations are performed using **vectorized operations**, which leverage the highly optimized C and Fortran backends of Pandas and NumPy.

- **Coefficient Mapping**: To apply business rule coefficients (e.g., for package type or delivery zone), the pipeline uses `pandas.Series.map()`. This method uses an efficient hash-table-based lookup, which is significantly faster than using an `apply()` function with a lambda or a Python `for` loop with `if/else` conditions.

  ```
  # From domain.py - Fast, vectorized lookup
  df['Package_Type'].map(PACKAGE_TYPE_COEFFS)
  ```

- **Conditional Categorization**: For logic that depends on multiple conditions, such as determining peak hours, `numpy.select()` is used. This function evaluates a

list of conditions and returns an array of corresponding choices, performing the entire operation in compiled C code.

```
# From domain.py - Highly optimized conditional logic
conditions = [...]
choices = [...]
df['Peak_Hour_Type'] = np.select(conditions, choices, default='Off-Peak')
```

- **Array-Based Mathematics**: All numerical calculations are performed on entire DataFrame columns at once. This ensures that the computations are executed as fast, parallelizable array operations rather than slow, one-by-one calculations in the Python interpreter.

3.2. Efficient I/O (Input/Output) Operations

Data loading and writing are common performance bottlenecks. The pipeline addresses this with the following strategies:

- **Bulk Database Inserts**: When generating the source SQLite database, the script first prepares all records in a Python list. It then uses `cursor.executemany()` to insert all records in a **single transaction**. This dramatically reduces disk I/O and transaction overhead compared to inserting and committing each record individually.
- **Optimized File Formats**: The pipeline supports writing to **Apache Parquet**. Parquet is a columnar storage format that offers superior compression and read performance for analytical workloads, as queries can read only the necessary columns instead of scanning entire rows.
- **Memory-Conscious Excel Export**: Writing large Excel files can consume vast amounts of RAM. To prevent this, the pipeline uses the `xlsxwriter` engine with the `{'constant_memory': True}` option. This mode streams data to temporary files on disk instead of building the entire file in memory, allowing for the creation of massive `.xlsx` files without risk of crashing.

4. Development and Deployment Workflow

The project is equipped with a modern toolchain to ensure a consistent, reproducible, and reliable development workflow.

4.1. Environment Management with Conda

The `environment.yml` file allows any developer to create a perfectly replicated Python environment with the exact dependencies used in the project. This eliminates "it works on my machine" problems and ensures consistency across all development and testing stages.

```
# Create and activate the identical environment
conda env create -f environment.yml
conda activate pysupercourier
```

4.2. Containerization with Docker

For ultimate portability and isolation, the project is fully containerized using Docker. The `Dockerfile` and `docker-compose.yml` files define a self-contained environment that includes the OS, Python, all dependencies, and the application code.

- `Dockerfile`: Uses a `miniconda3` base image and creates the Conda environment from the `environment.yml` file.
- `docker-compose.yml`: Manages the container's lifecycle, handles interactive sessions (`tty: true`), and maps the local `output_files` directory to the container. This ensures that any data generated inside the container is immediately available on the host machine.

This setup guarantees that the pipeline will run identically on any machine with Docker installed, regardless of the host operating system or local configuration.

4.3. Automated Testing with Pytest

The `tests/` directory contains a comprehensive test suite using the `pytest` framework.

- `test_pipeline.py`: Includes unit tests for core business logic (`domain.py`) and integration tests for data generation, transformation, and loading. It uses fixtures to create isolated, temporary environments for each test.
- `benchmark_tests.py`: A script that automates performance testing by running the pipeline with increasingly large datasets, capturing execution times for scalability analysis.
- `pytest.ini`: Configures `pytest` to automatically discover and run tests with verbose output.

This testing strategy ensures code correctness, prevents regressions, and validates the performance of the pipeline under various conditions.