ETL Project Implementation Report

Lorran Caetano Machado Lopes

March 30, 2025

1 Introduction

This report describes the implementation of an ETL (Extract, Transform, Load) project developed in OCaml. The project processes order data from two sources (Orders and Order Items), applies transformations, and generates analytical summaries. The ETL pipeline handles filtering, joining, and aggregation operations to create business insights, such as total order amounts and period-based averages.

The solution follows functional programming principles with an emphasis on type safety, separation of pure and impure functions, and testing.

2 Project Overview

2.1 Data Model

The project works with two main data tables represented in the following Entity-Relationship Model:

Order	**OrderItem**
PK id: int	PK FK order_id: int
client_id: int	PK product_id: int
order_date: varchar(14) (ISO 8601)	quantity: int
status: varchar(15)	price: float
origin: varchar(1)	tax: float (%)

The relationships between these entities are:

- One Order can have multiple OrderItems (one-to-many relationship)
- OrderItems are linked to Orders through the order_id foreign key

2.2 Project Requirements

The project needed to fulfill the following core requirements:

1. Implement the ETL process in OCaml
2. Use functional programming techniques (map, reduce, filter)
3. Separate pure functions from impure functions
4. Load data from input files into Record structures
5. Implement helper functions for data conversion
6. Process the data through joining, filtering, and aggregation
7. Generate output summaries with calculated totals
8. Create period-based summaries grouped by month and year
9. Implement comprehensive tests for all pure functions

3 Project Structure

The project is organized using the Dune build system with the following structure:

- **/workspaces/ETL/etl/** - Main project directory
  - **bin/** - Contains the executable components
    - **main.ml** - CLI entry point with command-line argument parsing
    - **dune** - Build configuration for the executable
  - **lib/** - Core library modules
    - **types.ml** - Data type definitions (pure)
    - **helper.ml** - Conversion and utility functions (pure)
    - **filter.ml** - Data filtering functions (pure)
    - **transform.ml** - Data transformation functions (pure)
    - **reader.ml** - File and URL reading functions (impure)
    - **writer.ml** - File output functions (impure)
    - **db.ml** - Database operations (impure)
    - **etl.ml** - Main orchestration module
    - **etl.mli** - Interface definition
    - **dune** - Build configuration for the library
  - **test/** - Test files for verifying functionality
    - **test_etl.ml** - Main test file
    - **test_helper.ml** - Helper module tests
    - **test_filter.ml** - Filter module tests
    - **test_transform.ml** - Transform module tests
    - **dune** - Build configuration for tests
  - **data/** - Sample data files for testing and development
    - **order.csv** - Sample order data
    - **order_item.csv** - Sample order item data
    - **order_summary.csv** - Generated output
    - **order_summary.db** - SQLite database with results
    - **complete_physical_orders.csv** - Sample filtered data
  - **dune-project** - Project configuration for the Dune build system
  - **etl.opam** - OCaml package manager configuration

This organization has a clear separation between:

- Pure functions for data processing (in helper.ml, filter.ml, transform.ml)
- Impure I/O functions (in reader.ml, writer.ml, db.ml)
- Command-line interface (in bin/main.ml)
- Comprehensive tests (in the test/ directory)

4 Implementation Details

4.1 Core Data Types

The foundation of the project is a set of OCaml record types defined in the Types module:

- **order**: Represents an order with fields for id, client_id, order_date, status, and origin.
- **order_item**: Represents an item in an order with fields for order_id, product_id, quantity, price, and tax.
- **order_summary**: Contains aggregated data for an order with fields for order_id, total_amount, and total_taxes.
- **period_summary**: Represents time-based aggregations with fields for year, month, avg_revenue, avg_taxes, and total_orders.
- **filter_params**: Encapsulates filtering parameters with optional status and origin fields.

These types ensure type safety throughout the application and provide clear interfaces between different parts of the system.

4.2 Pure Functions

The project implements several modules containing pure functions (no side effects, deterministic):

4.2.1 Helper Module

The Helper module provides functions for data conversion and validation:

- **string_to_int**: Converts strings to integers with proper error handling for invalid inputs.
- **string_to_float**: Converts strings to floats with error handling.
- **row_to_order**: Transforms a CSV row (list of strings) into an order record, validating the input format.
- **row_to_order_item**: Transforms a CSV row into an order item record with validation.
- **csv_to_orders**: Converts multiple CSV rows to a list of order records using map.
- **csv_to_order_items**: Converts multiple CSV rows to a list of order item records.

4.2.2 Filter Module

The Filter module implements functions to filter orders based on specific criteria:

- **filter_by_params**: Takes a list of orders and filter parameters (status and origin), returning a filtered list of orders that match the specified criteria. The function handles both specified and unspecified parameters through pattern matching on option types.

4.2.3 Transform Module

The Transform module contains the core data transformation logic:

- **joined_record**: An intermediate record type that combines relevant fields from orders and items for processing.
- **join_orders_and_items**: Joins orders with their corresponding items, calculating the amount and tax for each combination. The function iterates through orders and items, matching on order_id, and produces a list of joined records.

- **create_order_summary_from_joined**: Takes a list of joined records for the same order and creates a summary with total amounts and taxes. The function uses fold_left to accumulate values from all items.
- **transform_with_join**: Combines filtering and joining operations. It takes optional status and origin parameters, filters orders accordingly, joins with items, groups by order_id, and creates summaries for each order.
- **extract_year_month**: Parses a date string to extract year and month components with error handling.
- **calculate_period_summaries**: Groups order summaries by time periods (year and month) and calculates averages. The function creates a mapping from order_id to date, groups orders by period, and computes average metrics for each period.

4.3 Impure Functions

The project also contains several modules with impure functions (with side effects, I/O operations):

4.3.1 Reader Module

The Reader module handles file and network I/O for input:

- **read_csv_file**: Reads a CSV file from the local file system.
- **download_url**: Downloads content from a remote URL.
- **read_csv_from_string**: Parses CSV data from a string.
- **read_csv_from_url**: Combines URL downloading and CSV parsing.
- **read_csv_file_or_url**: Automatically determines if the source is a file or URL.
- **read_orders** and **read_order_items**: Specialized functions for reading the specific data files.

4.3.2 Writer Module

The Writer module handles file output:

- **summary_to_row** and **summary_to_csv**: Convert summaries to CSV format.
- **period_summary_to_csv**: Converts period summaries to CSV format.
- **write_summaries**: Writes order summaries to a CSV file.
- **write_period_summaries**: Writes period summaries to a CSV file.

4.3.3 DB Module

The DB module handles SQLite database operations:

- **init_db**: Creates the necessary database tables.
- **insert_summary**: Inserts a single order summary into the database.
- **save_summaries_to_db**: Saves multiple summaries in a transaction.

4.3.4 Main Orchestration

The **run_etl** function in etl.ml orchestrates the entire ETL process:

- Reads order and order item data
- Converts CSV rows to record structures
- Applies transformations and filtering
- Writes results to CSV files
- Optionally generates period summaries
- Optionally saves to a SQLite database

5 ETL Workflow

The complete ETL workflow operates as follows:

5.1 Extract

The extraction phase reads the input data from CSV files:

- Orders data is loaded from the first input file or URL using the Reader module
- Order Items data is loaded from the second input file or URL
- The CSV data is parsed into OCaml record structures using the Helper module's conversion functions

5.2 Transform

The transformation phase applies the following operations:

- Filtering orders based on specified parameters (status and/or origin) using the Filter module
- Joining orders with their corresponding items using the Transform module
- Calculating order summaries with total amounts and taxes
- Creating period-based summaries with average metrics

5.3 Load

The loading phase outputs the processed data:

- Writing order summaries to a CSV file using the Writer module
- Optionally writing period summaries to a CSV file
- Optionally storing results in a SQLite database using the DB module

6 Testing

Comprehensive tests were implemented for all pure functions using the OUnit2 testing framework.

- **test_helper.ml**: Tests focused exclusively on the Helper module functions
- **test_filter.ml**: Tests focused exclusively on the Filter module functions
- **test_transform.ml**: Tests focused exclusively on the Transform module functions
- **test_etl.ml**: Tests covering functions from multiple modules (Helper, Transform, and Filter).

Each test function follows a consistent pattern of setting up test data, calling the function under test, and verifying the results through assertions.

7 Declaration of AI Usage

In the development of this project and the preparation of this report, Generative AI tools were used for:

- Assistance with code structure and organization
- Guidance on OCaml
- Documenting functions
- Creating tests
- Help with formatting and structuring the report

All code was written, tested, and validated by the student, with AI serving as a programming assistant and advisor.