Relatório de Desenvolvimento do Projeto DataDuner

1. Introdução

Este relatório documenta o processo de desenvolvimento do DataDuner, um projeto de ETL (Extract, Transform, Load) implementado utilizando princípios de programação funcional em OCaml. O objetivo é servir como um roteiro detalhado para quem desejar recriar ou estender o projeto no futuro.

2. Uso de IA Generativa

Declaro que neste projeto foi utilizada IA Generativa (GitHub Copilot) para auxiliar nas seguintes tarefas:

- Sugestão e aprimoramento da documentação de código
- Refatoração para melhorar a separação entre código puro e impuro
- Correção de erros na implementação da conexão com o banco de dados
- Criação de arquivos de teste para funções puras
- Melhorias na estrutura do README

A arquitetura geral, algoritmos principais e decisões de design foram concebidos independentemente, utilizando a IA como ferramenta de apoio ao desenvolvimento.

3. Planejamento e Arquitetura

3.1. Requisitos Iniciais

O projeto foi concebido com os seguintes requisitos:

- Implementação em OCaml utilizando operações de map, reduce e filter
- Separação clara entre funções puras e impuras
- Estruturas de dados baseadas em records
- Funções auxiliares para carregamento de dados

3.2. Requisitos Opcionais e Caprichosos

- Extração de dados via HTTP
- Armazenamento em SQLite
- Implementação de inner join
- Capacidade de agregação mensal
- Sistema de logging

3.3. Decisão Arquitetural

Para conseguir trabalhar com maior facilidade, foi adotada uma arquitetura modular com separação clara de responsabilidades:

- records.ml: Definição de tipos de dados fundamentais
- ex.ml: Extração e parsing de dados (funções puras)
- tr.ml: Transformação de dados (funções puras)
- fe.ml: Operações de rede e acesso externo (funções impuras)
- sq.ml: Operações de banco de dados (funções impuras)
- logger.ml: Sistema de logging (funções impuras)
- main.ml: Orquestração do fluxo ETL

4. Implementação

A seguir, segue a implementação dos arquivos na ordem em que foram criados.

4.1. Definição de Tipos (records.ml)

Como primeiro passo para a extração de dados, foi necessário criar os Records que seriam utilizados para poder utilizar esses dados nas demais funções.

Foram implementados, inicialmente, os seguintes tipos de dados:

- origin: “Enum” para origem do pedido (Physical | Online)
- order: Registro com informações do pedido
- item: Registro com informações do item
- joined: Registro combinado de pedido e item

Posteriormente, ao longo do desenvolvimento, os seguintes Records foram necessários:

- output: Registro de saída com valores agregados
- order_total: Registro para totais por pedido
- monthly_mean: Registro para médias mensais

Cada tipo foi documentado com comentários de docstring para clareza.

4.2. Extração de Dados (ex.ml)

Em seguida, é necessário montar uma maneira de extrair os dados do CSV que vêm como strings e depois como Rows, então este módulo implementa funções puras para parsing de dados CSV:

- Funções para parsing de tipos primitivos (parse_int, parse_float, etc.)
- Funções para extração de campos de linhas CSV (extract_field)
- Parsing completo de registros (parse_row_order, parse_row_item)

O design utiliza o tipo Result para representação de erros, permitindo composição funcional segura através do operador let*.

4.3. Aquisição via HTTP (fe.ml)

Visando já o requisito opcional de fazer a extração a partir de um arquivo HTTP estático, foi criado esse módulo, que contém funções impuras para acesso a dados externos:

- http_get: Obtenção de dados via HTTP
- parse_csv: Parsing de string CSV para estrutura de dados
- fetch_csv_data: Combinação de obtenção e parsing
- fetch_orders e fetch_items: Funções específicas para obtenção de pedidos e itens em Records a partir das funções construídas no módulo anterior

Foi utilizada a biblioteca Cohttp com Lwt para operações assíncronas de rede.

4.4. Transformação de Dados (tr.ml)

Aqui implementamos funções puras de transformação de dados:

- filter_by_status: Filtra pedidos por status e origem
- create_joined_record: Cria um registro unindo pedido e item
- inner_join: Implementa inner join entre pedidos e itens
- group_by: Agrupa registros por chave
- calculate_totals: Calcula totais de valores em uma lista
- calculate_means: Calcula médias de valores em uma lista
- month_year_key: Extrai chave ano/mês de um registro
- group_by_to_order_totals: Agrupa e calcula totais
- group_by_to_means: Agrupa e calcula médias

Foram aplicadas operações funcionais como map, filter, fold_left e filter_map para manter o código limpo e expressivo. Também, foram utilizadas Helper functions para poder diminuir o número de linhas de código e aumentar a modularização

4.5. Banco de Dados (sq.ml)

Implementamos operações de banco de dados SQLite:

- Criação de tabelas
  (create_order_totals_table, create_monthly_means_table)
- Inserção de dados (insert_order_total, insert_monthly_mean)
- Leitura de dados (get_all_order_totals, get_all_monthly_means)
- Função utilitária para executar várias queries em sequência (iter_queries)
- Wrapper para conexão com tratamento de erros (with_db)

Utilizamos ppx_rapper para gerar código seguro de acesso ao banco.

4.6. Sistema de Logging (logger.ml)

Criamos um sistema simples de logging:

- log_info e log_error: Funções básicas de logging
- Funções específicas para logging de diferentes tipos de dados
- Integração com Caqti para logging de erros de banco de dados

4.7. Módulo Principal (main.ml)

O módulo principal orquestra o fluxo ETL:

- Extração de dados usando Fe.fetch_orders e Fe.fetch_items
- Parsing de argumentos da linha de comando
- Transformação de dados usando as funções puras de Tr
- Logging de resultados com Logger
- Salvamento em banco de dados usando Sq

Procuramos manter a parte pura do código separada da impura através de composição funcional.

5. Documentação e Considerações Finais

A documentação inclui comentários inline e um README básico que explica a visão geral do projeto, a arquitetura modular e as instruções de instalação e uso. Apesar da pressa, todos os módulos principais foram comentados com docstrings que descrevem parâmetros, retorno e lógica das funções.

Lições Aprendidas

- A separação entre funções puras e impuras é fundamental, embora a implementação apresente áreas que podem ser refinadas.

- O uso de operações funcionais facilita o processamento dos dados, mesmo que o tratamento de erros ainda precise de melhorias.
- A modularidade permite que o sistema seja estendido, apesar do desenvolvimento rápido e improvisado.

Melhorias Futuras

- Incrementar o tratamento de erros e adicionar testes unitários, especialmente para as funções impuras.
- Refinar a interface de linha de comando e implementar suporte para outros formatos além de CSV.
- Melhorar a robustez do sistema de logging e otimizar as operações com o banco de dados.

Conclusão

Este relatório descreve o desenvolvimento do DataDuner de forma rápida, destacando os pontos essenciais do código e a estrutura modular do projeto. Mesmo com a implementação feita às pressas, o sistema demonstra a aplicação prática dos princípios de programação funcional em OCaml e fornece uma base sólida para futuras melhorias e extensões.