Sailco.ipynb Code Documentation

MOSDEX to SQLAlchemy Database Converter

Executive Summary

This Jupyter notebook implements a sophisticated Python-based system that reads MOSDEX (Mathematical Optimization Solver Data EXchange) JSON files and automatically transforms them into SQLAlchemy database tables. The implementation demonstrates the Sailco inventory optimization problem, showcasing how optimization model data can be managed using modern data engineering practices [1] [2].

1. Introduction to MOSDEX

1.1 What is MOSDEX?

MOSDEX (Mathematical Optimization Solver Data EXchange) is a modern standard for managing data interaction with optimization solvers. Developed to address the computational challenges of large-scale optimization problems, MOSDEX provides a standardized JSON-based format that enables [1] [2]:

- **Solver Independence**: Works with multiple optimization solvers (CPLEX, Gurobi, etc.)
- **Language Neutrality**: Supports multiple programming languages (Python, Java, Julia, C++)
- **Data-Model Separation**: Clean separation between optimization model structure and instance data
- **Relational Structure**: Uses familiar database concepts (tables, schemas, queries)
- **SQL Integration**: Leverages SQL for data transformations

1.2 Why MOSDEX Matters

Traditional optimization instance formats (MPS, LP) represent the end product of upstream data transformations, which are often ad-hoc and undocumented. MOSDEX addresses this by providing [1] [2]:

1. A standardized way to represent both model structure and data
2. Support for the entire optimization workflow (data preparation → solving → results retrieval)
3. Human-readable format for debugging and verification
4. Efficient handling of large datasets typical in modern analytics

2. The Sailco Problem

2.1 Problem Description

The Sailco problem is a classic inventory optimization case study involving a company that manufactures sailboats [3]. The company must:

- Meet demand forecasts over multiple time periods
- Decide production levels (regular and overtime)
- Manage inventory levels
- Minimize total costs (production + inventory holding)

2.2 Mathematical Formulation

The problem includes:

Decision Variables:

- RP_t: Regular production in period t
- EP_t: Extra (overtime) production in period t
- inv_t: Inventory level at end of period t

Objective: Minimize total cost across all periods

Constraints:

- Production capacity limits
- Inventory balance equations
- Non-negativity constraints

3. Notebook Architecture

3.1 Overall Structure

The notebook consists of 5 cells implementing a complete MOSDEX processing pipeline:

1. **Cell 1**: MOSDEX file validation and database initialization
2. **Cell 2**: Dynamic table creation and data population
3. **Cells 3-5**: Template code for object-oriented refactoring (not executed)

3.2 Data Flow

MOSDEX JSON → Schema Validation → Model Extraction → Dynamic Table Creation → Data Population → Query Execution → Database Tables

4. Cell 1: MOSDEX File Validation and Database Initialization

4.1 Library Imports

```
import json
import pprint
from jsonschema.validators import Draft7Validator
from sqlalchemy import create_engine, Integer, String, Double, ForeignKey
from sqlalchemy.orm import declarative_base, mapped_column
```

Purpose and Dependencies:

- **json**: Standard library for parsing JSON files; no installation required [4]
- **pprint**: Pretty-printer for formatting validation error messages
- **Draft7Validator**: From jsonschema package; validates JSON documents against JSON Schema Draft 7 specification [5] [6]
- **sqlalchemy**: Comprehensive ORM toolkit for database interaction [7] [8] [9]

Key Concepts:

The Draft7Validator implements JSON Schema Draft 7, which defines a vocabulary for describing JSON data structures and validation rules. It checks that the MOSDEX file conforms to the defined schema structure [5] [10].

4.2 MOSDEX File Loading and Validation

```
MOSDEX_SCHEMA_FILE = "MOSDEXSchemaV2-1.json"
MOSDEX_FILE = "sailco_2-1.json"

with open(MOSDEX_SCHEMA_FILE, "r") as f:
    schema = json.load(f)
validator = Draft7Validator(schema)

with open(MOSDEX_FILE, "r") as f:
    mosdex = json.load(f)

if not validator.is_valid(mosdex):
    print(f"File {MOSDEX_FILE} is not a valid Mosdex file.")
    pp = pprint.PrettyPrinter(indent=4)
    for error in sorted(validator.iter_errors(mosdex), key=str):
        print()
        pp.pprint(error.message)
else:
    print(f"File {MOSDEX_FILE} is a valid instance of schema {MOSDEX_SCHEMA_FILE}.")
```

Functionality:

1. **Schema Loading**: Loads the MOSDEX schema definition (MOSDEXSchemaV2-1.json) which specifies the structure that all MOSDEX files must follow [11]

2. **Validator Creation**: Instantiates a Draft7Validator with the loaded schema [5] [6]
3. **File Loading**: Loads the sailco problem instance (sailco_2-1.json)
4. **Validation**: Uses `validator.is_valid()` to check conformance [10]
5. **Error Reporting**: If validation fails, iterates through errors and pretty-prints them for debugging [12]

Schema Validation Benefits:

- Ensures data integrity before processing
- Catches structural errors early
- Provides detailed error messages for debugging
- Enforces consistency across different MOSDEX files

4.3 Model Extraction

```
model = {}
for module in mosdex['MODULES']:
    if module['KIND'] == 'MODEL':
        model = module
        break

print(f"Got handle to MODEL: {model['NAME']}")
print(f"The sections of the model are {list(model.keys())}")
print(f"\\tNAME: {model['NAME']}")
print(f"\\tCLASS: {model['CLASS']}")
print(f"\\tKIND: {model['KIND']}")
print(f"\\tTABLES: there are {len(model['TABLES'])} tables:")
for table in model['TABLES']:
    print(f"\\t\\t{table['NAME']:10s} \\t class/kind: {table['CLASS']}/{table['KIND']}")
```

MOSDEX Structure:

A MOSDEX file contains a `MODULES` array, where each module represents a logical component [3] [11]:

- **MODEL**: Contains the optimization model structure (variables, constraints, objective)
- **DATA**: Contains input data instances
- **SOLUTION**: Contains solver results
- **SCHEMA**: Defines data structures

Each module has attributes:

- `NAME`: Unique identifier
- `CLASS`: Classification (MODULE)
- `KIND`: Type of module (MODEL, DATA, SOLUTION)
- `TABLES`: Array of table definitions

Output Example:

```
Got handle to MODEL: sailco
The sections of the model are ['NAME', 'CLASS', 'KIND', 'HEADING', 'TABLES']
        NAME: sailco
        CLASS: MODULE
        KIND: MODEL
        TABLES: there are 10 tables:
                demands          class/kind: DATA/INPUT
                parameters       class/kind: DATA/INPUT
                RP               class/kind: VARIABLE/CONTINUOUS
                EP               class/kind: VARIABLE/CONTINUOUS
                inv              class/kind: VARIABLE/CONTINUOUS
                cost             class/kind: VARIABLE/CONTINUOUS
                ctinv            class/kind: CONSTRAINT/LINEAR
                ctinvMat         class/kind: MATRIX/LINEAR
                ctCost           class/kind: CONSTRAINT/LINEAR
                ctCostMat        class/kind: MATRIX/LINEAR
```

4.4 Database Engine Initialization

```
DB_ENGINE = 'sqlite:///:memory:'
# DB_ENGINE = "duckdb:///:memory:"
engine = create_engine(DB_ENGINE)

Base = declarative_base()
Base.metadata.drop_all(bind=engine)

print(f"Database engine {DB_ENGINE} created and all tables dropped.")
```

SQLAlchemy Engine Concepts [7] [8] [9]:

1. **Engine**: The starting point for SQLAlchemy applications; manages database connections and provides a source of database connectivity [7]
2. **Connection String Format**:
    - `dialect://[user:password@][host:port]/database`
    - `sqlite:///:memory:` creates an in-memory SQLite database
    - Alternative: `duckdb:///:memory:` for DuckDB (commented out)
3. **Declarative Base**: Creates a base class for declarative ORM models [7] [13]
    - All table classes will inherit from this base
    - Maintains metadata about all tables
    - Enables ORM features like relationships and queries
4. **Drop All Tables**: `Base.metadata.drop_all()` removes all existing tables for a clean start [7]
    - Useful for development and testing
    - Ensures fresh schema on each run

Database Engine Options:

Engine	Advantages	Disadvantages
SQLite	No setup required, portable, fast for small datasets	No sequence support, limited concurrency
DuckDB	Analytical queries, better performance on large datasets	Requires installation, sequence support issues
PostgreSQL	Full SQL support, production-ready, sequences supported	Requires server setup

5. Cell 2: Dynamic Table Creation and Data Population

5.1 Sequence Setup for Auto-Incrementing IDs

```
from sqlalchemy import Sequence
import sqlalchemy

user_id_seq = Sequence('user_id_seq')
```

Purpose: Create a sequence object for generating auto-incrementing primary keys [4].

Background on Sequences:

- **Sequences**: Database objects that generate unique sequential numbers
- **SQLite Limitation**: SQLite uses AUTOINCREMENT for primary keys and doesn't support explicit sequences [4]
- **DuckDB Workaround**: Requires explicit sequence definition for auto-incrementing columns
- **Error in Notebook**: The code fails when run with SQLite because sequences are not supported [4]

Recommended Fix:

```
# For SQLite, remove sequence and use autoincrement
table_attr = {
    '__tablename__': table_name,
    'id': mapped_column(Integer, primary_key=True, autoincrement=True)
}

# Or switch to PostgreSQL/DuckDB
DB_ENGINE = "postgresql://user:password@localhost/database"
```

5.2 Database Tables Dictionary Initialization

```
db_tables = {}
for key in ['variables', 'constraints', 'matrix']:
    db_tables[key] = {}
    db_tables[key]['table_instance_name'] = key
    db_tables[key]['table_class_name'] = model['NAME'].capitalize() + key.capitalize()
    table_attr = {
        '__tablename__': db_tables[key]['table_instance_name'],
        'id': mapped_column(Integer, user_id_seq, server_default=user_id_seq.next_value()
        'state': mapped_column(String)
    }
    db_tables[key]['table_class'] = type(db_tables[key]['table_class_name'], (Base,), tab
```

Purpose: Pre-create metadata storage for optimization model artifacts.

Data Structure:

The `db_tables` dictionary stores comprehensive metadata for each table:

```
db_tables = {
    'variables': {
        'table_instance_name': 'variables',
        'table_class_name': 'SailcoVariables',
        'table_class': &lt;dynamically created class&gt;
    },
    'constraints': {...},
    'matrix': {...}
}
```

Dynamic Type Creation:

The code uses Python's `type()` function to create classes at runtime [7] [9]:

```
type(name, bases, dict)
```

Where:

- `name`: String name of the class (e.g., 'SailcoVariables')
- `bases`: Tuple of base classes (e.g., `(Base,)`)
- `dict`: Dictionary of attributes (columns, methods)

This approach enables schema-driven programming where the database structure is entirely defined by the MOSDEX JSON file, not hardcoded.

5.3 Processing MOSDEX Tables

```
for table_ in model['TABLES']:
    # Record table metadata
    key = table_['NAME']
    db_tables[key] = {}
    db_tables[key]['table_instance_name'] = key
    db_tables[key]['table_class_name'] = model['NAME'].capitalize() + key.capitalize()

    # Store MOSDEX metadata
    db_tables[key]['mosdex_json'] = table_
    db_tables[key]['mosdex_schema'] = table_['SCHEMA']
    db_tables[key]['mosdex_class'] = table_['CLASS']
    db_tables[key]['mosdex_kind'] = table_['KIND']
```

MOSDEX Table Structure [3] [11]:

Each table in the MOSDEX TABLES array has:

1. **NAME**: Unique identifier (e.g., "demands", "RP", "ctinv")
2. **CLASS**: Type of optimization artifact
    - DATA: Input data
    - VARIABLE: Decision variables
    - CONSTRAINT: Problem constraints
    - MATRIX: Constraint coefficient matrices
3. **KIND**: Subtype (e.g., INPUT, CONTINUOUS, LINEAR)
4. **SCHEMA**: Column definitions
    - NAME: Array of column names
    - KIND: Array of data types
    - KEYS: Primary key columns
    - FOREIGN_KEYS: Foreign key relationships
5. **INSTANCE** or **QUERY**: Data source
    - INSTANCE: Static data arrays
    - QUERY: SQL-like derivation rules

5.4 Column Type Mapping

```
table_name = db_tables[key]['table_instance_name']
table_schema = db_tables[key]['mosdex_schema']

table_attr = {
    '__tablename__': table_name,
    'id': mapped_column(Integer, user_id_seq, server_default=user_id_seq.next_value(), pr
}

if db_tables[key]['mosdex_class'] in ['VARIABLE', 'CONSTRAINT', 'MATRIX']:
    table_attr = {
        '__tablename__': table_name,
        'id': mapped_column(Integer, primary_key=True)
    }

# Apply the SCHEMA
for name, kind in zip(table_schema['NAME'], table_schema['KIND']):
    if kind == 'INTEGER':
        type_col = Integer
    elif kind == 'DOUBLE' or kind == 'DOUBLE_FUNCTION':
        type_col = Double
    elif kind == 'STRING':
        type_col = String
    else:
        print(f"Error, type {kind} is not supported. Detected in Table {table_name}")
        break
```

Type Mapping Logic:

MOSDEX Type	SQLAlchemy Type	Python Type	Usage
INTEGER	Integer	int	Periods, counts, indices
DOUBLE	Double	float	Costs, demands, coefficients
DOUBLE_FUNCTION	Double	float	Computed values
STRING	String	str	Names, labels, identifiers

Special Handling for Optimization Tables:

Variables, constraints, and matrix tables don't use auto-increment sequences because they have composite primary keys defined in the schema (e.g., period + variable name).

5.5 Primary and Foreign Key Relationships

```
if 'KEYS' in table_schema and name in table_schema['KEYS']:
    # Primary Key
    table_attr[name] = mapped_column(type_col, primary_key=True)
elif 'FOREIGN_KEYS' in table_schema and name in table_schema['FOREIGN_KEYS']:
    # Foreign Key
    f_key = table_schema['FOREIGN_KEYS'][name]
    table_attr[name] = mapped_column(type_col, ForeignKey(f_key))
else:
    table_attr[name] = mapped_column(type_col)
```

Relational Database Concepts [7] [8]:

1. **Primary Keys**: Uniquely identify each row in a table
    - Can be single column or composite (multiple columns)
    - Example: `period` in the `demands` table
    - Automatically indexed for fast lookups
2. **Foreign Keys**: Reference primary keys in other tables
    - Enforce referential integrity
    - Example: `period` in `RP` table references `demands.period`
    - Prevent orphaned records
    - Enable JOIN operations

Sailco Example Relationships:

```
demands (period [PK], demand)
    ↓ (referenced by)
RP (period [FK], Col [PK], lowerBound, upperBound, primalValue)
EP (period [FK], Col [PK], ...)
inv (period [FK], Col [PK], ...)
```

5.6 Table Class Creation

```
# Declarative instantiation of the table
db_tables[key]['table_class'] = type(
    db_tables[key]['table_class_name'],
    (Base,),
    table_attr
)

Base.metadata.create_all(engine)

print(f"Database tables created")
for table in Base.metadata.tables.keys():
    print(f"\\t{table}")
    print(f"\\t\\t{Base.metadata.tables[table].columns.keys()}")
```

SQLAlchemy Declarative Mapping [7] [8] [13]:

The declarative approach creates Python classes that map to database tables. Each class:

1. Inherits from the declarative `Base`
2. Has a `__tablename__` attribute specifying the table name
3. Contains `mapped_column()` definitions for each column
4. Automatically gets query capabilities

Resulting Table Structure (example for RP variable):

```
class SailcoRP(Base):
    __tablename__ = 'RP'
    id = mapped_column(Integer, primary_key=True)
    period = mapped_column(Integer, primary_key=True)
    Col = mapped_column(String, primary_key=True)
    lowerBound = mapped_column(Double)
    upperBound = mapped_column(Double)
    primalValue = mapped_column(Double)
```

`Base.metadata.create_all(engine)`:

This command creates all tables in the database that have been defined but don't yet exist [7]:

- Analyzes all classes inheriting from Base
- Generates CREATE TABLE SQL statements
- Executes them on the database

- Skips tables that already exist (when `checkfirst=True`, which is default)

5.7 INSTANCE Data Processing

```
from sqlalchemy import text, Table
from sqlalchemy.orm import Session
from prettytable import PrettyTable
import pandas as pd
import numpy as np

for key in db_tables.keys():
    if 'mosdex_json' not in db_tables[key]:
        continue

    if "INSTANCE" in db_tables[key]['mosdex_json']:
        # Get column names
        col_names = db_tables[key]['mosdex_schema']['NAME']

        # Create a dataframe from the INSTANCE arrays
        data_df = pd.DataFrame(
            np.vstack(db_tables[key]['mosdex_json']['INSTANCE']),
            columns=col_names
        )

        # Push the dataframe to the table
        with Session(engine) as session, session.begin():
            table_name = db_tables[key]['table_instance_name']
            data_df.to_sql(
                name=table_name,
                con=session.connection(),
                if_exists='append',
                index=False
            )
            session.flush()
            stmt = "select * from " + table_name

            rows = session.execute(text(stmt))
            pretty_table = PrettyTable()
            pretty_table.field_names = data_df.columns
            for row in rows:
                pretty_table.add_row(row[1:])
            print(pretty_table)
```

INSTANCE Data Format [3] [11]:

In MOSDEX, INSTANCE provides static data as arrays of arrays:

```
{
  "NAME": "demands",
  "CLASS": "DATA",
  "KIND": "INPUT",
  "SCHEMA": {
    "NAME": ["period", "demand"],
    "KIND": ["INTEGER", "DOUBLE"],
    "KEYS": ["period"]
  },
  "INSTANCE": [
    [1, 40.0],
    [2, 60.0],
    [3, 75.0],
    [4, 25.0]
  ]
}
```

Data Processing Pipeline:

1. **NumPy vstack**: Stacks INSTANCE arrays vertically into a 2D array
2. **pandas DataFrame**: Creates structured data with column names
3. **Session Context**: Manages database transaction [7] [8]
   - `with Session(engine) as session, session.begin()` creates a transaction
   - Automatically commits on success, rolls back on error
4. **to_sql()**: Bulk inserts DataFrame rows into table
   - `if_exists='append'`: Adds to existing data
   - `index=False`: Don't insert DataFrame index as column
5. **session.flush()**: Sends pending changes to database
6. **PrettyTable**: Formats output for display

Benefits of Bulk Loading:

- Much faster than row-by-row insertion
- Single database transaction
- Efficient memory usage
- Automatic type conversion

5.8 QUERY Directive Processing

```
for key in db_tables.keys():
    if 'mosdex_json' not in db_tables[key]:
        continue
    if 'QUERY' in db_tables[key]['mosdex_json']:
        for statement in db_tables[key]['mosdex_json']['QUERY']:
            insert_array = db_tables[key]['mosdex_schema']['NAME']
            select_array = statement['SELECT']
            from_array = statement['FROM']

            insert_stmt = "INSERT INTO " + key + '(' + ",".join(insert_array) + ')'
            select_stmt = "SELECT " + ",".join(select_array)
            from_stmt = "FROM " + ",".join(from_array)
            stmt = insert_stmt + ' ' + select_stmt + " " + from_stmt

            if "JOIN" in statement:
                join_array = statement['JOIN']
                join_stmt = " JOIN " + " JOIN ".join(join_array)
                stmt = stmt + join_stmt

            if "WHERE" in statement:
                where_array = statement['WHERE']
                stmt = stmt + " WHERE " + " ".join(where_array)

            with Session(engine) as session, session.begin():
                session.execute(text(stmt))
```

QUERY Directive Structure [3] [2]:

QUERY directives specify SQL-like operations to derive table data from other tables:

```
{
  "QUERY": [
    {
      "SELECT": [
        "demands.period AS period",
        "CONCAT('RP_', period) AS Col",
        "0.0 AS lowerBound",
        "parameters.capacity AS upperBound",
        "NULL AS primalValue"
      ],
      "FROM": ["demands", "parameters"]
    }
  ]
}
```

SQL Statement Construction:

The code builds SQL statements dynamically:

```
# Example output:
INSERT INTO RP(period, Col, lowerBound, upperBound, primalValue)
SELECT demands.period AS period,
       CONCAT('RP_', period) AS Col,
       0.0 AS lowerBound,
       parameters.capacity AS upperBound,
       NULL AS primalValue
FROM demands, parameters
```

Advanced Features:

1. **JOIN Clauses**:

```
"JOIN": [
  "ctinv ON ctinv.period = demands.period",
  "RP ON RP.period = demands.period"
]
```

2. **WHERE Clauses**:

```
"WHERE": ["demands.period = 1"]
```

Conditional Data Derivation:

The Sailco model uses WHERE clauses to handle boundary conditions:

- First period (t=1): Different constraint because no prior inventory
- Subsequent periods (t>1): Include lagged inventory term

This enables complex, multi-row QUERY directives:

```
"QUERY": [
  {
    "SELECT": [...],
    "FROM": [...],
    "WHERE": ["demands.period = 1"]
  },
  {
    "SELECT": [...],
    "FROM": [...],
    "WHERE": ["demands.period &gt; 1"]
  }
]
```

5.9 Constraint and Matrix Processing

```
# Process CONSTRAINTS
for key in db_tables.keys():
    if 'mosdex_json' not in db_tables[key]:
        continue
    if "CONSTRAINT" == db_tables[key]['mosdex_json']['CLASS']:
        # ... same QUERY processing as above ...

# Process MATRIX
for key in db_tables.keys():
    if 'mosdex_json' not in db_tables[key]:
        continue
    if "MATRIX" == db_tables[key]['mosdex_json']['CLASS']:
        # ... same QUERY processing as above ...
```

Optimization Model Structure [2]:

MOSDEX separates optimization models into distinct components:

1. **Variables**: Decision variables (what to optimize)
    - RP: Regular production quantity per period
    - EP: Extra production quantity per period
    - inv: Inventory level per period

2. **Constraints**: Feasibility requirements
    - ctinv: Inventory balance (production + prior inventory = demand + ending inventory)
    - ctCost: Cost calculation
3. **Matrix**: Constraint coefficients
    - Maps which variables appear in which constraints
    - Specifies coefficients (typically +1, -1, or parameter values)
    - ctinvMat: Coefficients for inventory constraints
    - ctCostMat: Coefficients for cost constraints

Matrix Table Example:

The ctinvMat table stores constraint coefficients in relational format:

period	row	RPCol	RPCoeff	EPCol	EPCoeff	invCol	invCoeff	LaginvCol	LaginvCoeff
1	ctinv_1	RP_1	1.0	EP_1	1.0	inv_1	-1.0	NULL	NULL
2	ctinv_2	RP_2	1.0	EP_2	1.0	inv_2	-1.0	inv_1	1.0

This represents the constraint: $RP_2 + EP_2 + inv_1 - inv_2 = demand_2$

5.10 Result Display

```
table = Table('ctInvMat', Base.metadata, autoload_with=engine)

stmt = "SELECT period, row, invCol, invCoeff, LaginvCol, LaginvCoeff FROM ctInvMat"

with engine.connect() as connection:
    result = connection.execute(text(stmt))

pretty_table = PrettyTable()
pretty_table.field_names = ['period', 'row', 'col1', 'coeff1', 'col2', 'coeff2']

for row in result:
    pretty_table.add_row(row)

print(pretty_table)
```

Table Reflection [7]:

SQLAlchemy can "reflect" existing database tables:

```
Table('ctInvMat', Base.metadata, autoload_with=engine)
```

This:

- Queries database schema
- Discovers column names and types

- Creates Table object without explicit definition
- Useful for working with existing databases

Connection vs Session [7] [8]:

- **Connection**: Lower-level, executes SQL directly
- **Session**: Higher-level, manages ORM objects and transactions

Both approaches work; Sessions provide more features for ORM usage.

6. Cells 3-5: Template Code (Not Executed)

These cells contain refactored code showing best practices:

6.1 MosdexTable Class (Cell 3)

```
class MosdexTable(Base):
    def process_instance(self, instance_data, engine):
        import pandas as pd
        from sqlalchemy.orm import Session

        col_names = self.table_schema['NAME']
        data_df = pd.DataFrame(instance_data, columns=col_names)

        with Session(engine) as session:
            data_df.to_sql(
                name=self.__tablename__,
                con=session.connection(),
                if_exists="append",
                index=False
            )
            session.commit()
```

Object-Oriented Design:

This template demonstrates:

- Encapsulation of INSTANCE processing logic
- Reusable method for any MOSDEX table
- Clean separation of concerns
- Easier testing and maintenance

6.2 Database Initialization Function (Cell 4)

```
def initialize_database(engine_url: str = "sqlite:///:memory:"):
    from sqlalchemy import create_engine

    engine = create_engine(engine_url)
    Base.metadata.drop_all(bind=engine)
    Base.metadata.create_all(bind=engine)
    print(f"Database initialized at {engine_url}")
    return engine
```

Configuration Management:

Benefits:

- Single point of configuration
- Easy to switch database backends
- Default value for development
- Returns engine for further use

6.3 Query Processing Method (Cell 5)

```
class MosdexTable(Base):
    def process_query(self, query_data, engine):
        from sqlalchemy.orm import Session
        from sqlalchemy.sql import text

        for statement in query_data:
            stmt = (
                f"INSERT INTO {self.__tablename__} "
                f"({','.join(self.table_schema['NAME'])}) "
                f"SELECT {','.join(statement['SELECT'])} "
                f"FROM {','.join(statement['FROM'])}"
            )

            if "JOIN" in statement:
                stmt += f" JOIN {' JOIN '.join(statement['JOIN'])}"

            if "WHERE" in statement:
                stmt += f" WHERE {' '.join(statement['WHERE'])}"

            with Session(engine) as session:
                session.execute(text(stmt))
                session.commit()
```

Modular Architecture:

This refactoring enables:

- One method per responsibility
- Easier unit testing
- Better error handling
- Code reuse across different MOSDEX files

7. Key Design Patterns

7.1 Dynamic Type Creation

Pattern: Metaprogramming using `type()`

Implementation:

```
SailcoRP = type('SailcoRP', (Base,), {
    '__tablename__': 'RP',
    'period': mapped_column(Integer, primary_key=True),
    'Col': mapped_column(String, primary_key=True),
    'lowerBound': mapped_column(Double),
    'upperBound': mapped_column(Double)
})
```

Benefits:

- Schema entirely data-driven
- No hardcoded table definitions
- Supports any MOSDEX file structure
- Enables generic processing framework

7.2 Schema-Driven Development

Pattern: Configuration over code

Key Principle: The MOSDEX JSON file is the single source of truth for:

- Table structures
- Column types
- Relationships
- Data sources
- Transformations

Advantages:

- Changes to model don't require code changes
- Same code works for any MOSDEX-compliant model
- Easy to version control models
- Self-documenting structure

7.3 Declarative ORM

Pattern: Map Python classes to database tables [7] [8] [13]

Benefits:

- Clean, Pythonic API
- Automatic query generation
- Type checking
- Relationship traversal
- Transaction management

7.4 Bulk Data Operations

Pattern: Batch processing using pandas [4]

Implementation:

```
data_df = pd.DataFrame(instance_data, columns=col_names)
data_df.to_sql(name=table_name, con=connection, if_exists='append')
```

Performance:

- 100-1000x faster than row-by-row insertion
- Single database transaction
- Efficient memory usage
- Built-in type conversion

8. Common Issues and Solutions

8.1 Sequence Error

Problem:

```
NotImplementedError: Dialect 'sqlite' does not support sequence increments.
```

Root Cause: SQLite uses AUTOINCREMENT, not sequences [4].

Solutions:

Option 1: Remove sequence usage

```
table_attr = {
    '__tablename__': table_name,
    'id': mapped_column(Integer, primary_key=True, autoincrement=True)
}
```

Option 2: Switch to PostgreSQL

```
DB_ENGINE = "postgresql://username:password@localhost:5432/sailco"
```

Option 3: Use DuckDB (if compatible)

```
DB_ENGINE = "duckdb:///sailco.db"
```

8.2 Memory Database Persistence

Problem: Data lost when kernel restarts

Solution: Use file-based database

```
DB_ENGINE = "sqlite:///sailco.db"  # Creates sailco.db file
```

8.3 Foreign Key Validation

Problem: Foreign key constraint violations

Causes:

- Referenced table not created yet
- Referenced data not inserted yet
- Mismatched data types

Solutions:

1. Ensure proper table creation order
2. Use `Base.metadata.create_all()` (handles dependencies automatically)
3. Disable foreign keys during bulk load, re-enable after:

```
session.execute(text("PRAGMA foreign_keys=OFF"))
# ... bulk load ...
session.execute(text("PRAGMA foreign_keys=ON"))
```

8.4 Type Mismatches

Problem: "DOUBLE_FUNCTION" not recognized

Solution: Add to type mapping

```
if kind in ['DOUBLE', 'DOUBLE_FUNCTION']:
    type_col = Double
```

9. Dependencies and Environment

9.1 Required Packages

```
pip install sqlalchemy jsonschema pandas numpy prettytable
```

9.2 Optional Packages

```
# For DuckDB support
pip install duckdb duckdb-engine

# For PostgreSQL support
pip install psycopg2-binary

# For visualization
pip install matplotlib seaborn
```

9.3 Python Version

Requires Python 3.8+ for:

- SQLAlchemy 2.0 features
- Type hints
- F-string syntax

10. Extending the Framework

10.1 Adding Custom Validators

```
def validate_positive(value):
    if value &lt;= 0:
        raise ValidationError("Value must be positive")

# Register custom validator
Draft7Validator.VALIDATORS["validate_positive"] = \
    lambda validator, validate_positive, instance, schema: \
    validate_positive(instance) if instance else None
```

10.2 Supporting Additional Data Types

```
type_map = {
    'INTEGER': Integer,
    'DOUBLE': Double,
    'DOUBLE_FUNCTION': Double,
    'STRING': String,
    'BOOLEAN': Boolean,  # Add new type
    'DATE': Date,        # Add new type
    'DATETIME': DateTime # Add new type
}
```

10.3 Adding Solver Integration

```
from docplex.mp.model import Model

def export_to_cplex(db_tables, engine):
    model = Model(name='sailco')

    # Load variables
    with Session(engine) as session:
        rp_vars = session.query(db_tables['RP']['table_class']).all()
        for var in rp_vars:
            model.continuous_var(
                lb=var.lowerBound,
                ub=var.upperBound,
                name=var.Col
            )

    # Add constraints...
    # Set objective...

    solution = model.solve()
    return solution
```

11. Performance Considerations

11.1 Optimization Strategies

For Large Datasets:

1. **Batch Size**: Tune `to_sql()` chunksize

```
data_df.to_sql(name=table_name, con=connection, chunksize=1000)
```

2. **Indexing**: Create indexes on foreign keys

```
Index('idx_period', 'period').create(engine)
```

3. **Bulk Operations**: Use bulk insert for ORM

```
session.bulk_insert_mappings(SailcoRP, data_dicts)
```

4. **Connection Pooling**: Configure pool size

```
engine = create_engine(DB_ENGINE, pool_size=10, max_overflow=20)
```

11.2 Memory Management

For Large MOSDEX Files:

1. **Streaming JSON**: Use ijson for large files

```
import ijson
with open(MOSDEX_FILE, 'rb') as f:
    for table in ijson.items(f, 'MODULES.item.TABLES.item'):
        process_table(table)
```

2. **Incremental Processing**: Process tables one at a time
3. **Garbage Collection**: Explicitly free memory

```
import gc
del data_df
gc.collect()
```

12. Testing Strategies

12.1 Unit Tests

```
import unittest

class TestMosdexProcessor(unittest.TestCase):
    def setUp(self):
        self.engine = initialize_database("sqlite:///:memory:")

    def test_table_creation(self):
        # Test that all tables are created
        tables = Base.metadata.tables.keys()
        self.assertIn('demands', tables)
        self.assertIn('RP', tables)

    def test_instance_loading(self):
        # Test data loading
        with Session(self.engine) as session:
            demands = session.query(Demands).count()
            self.assertEqual(demands, 4)  # Sailco has 4 periods
```

12.2 Integration Tests

```
def test_full_pipeline():
    # Load MOSDEX
    mosdex = load_mosdex("sailco_2-1.json")

    # Process
    engine = initialize_database()
    process_mosdex(mosdex, engine)
    # Verify
    with Session(engine) as session:
        # Check that derived tables have correct data
        rp_count = session.query(RP).count()
        assert rp_count == 4  # One per period
```

13. Conclusion

13.1 Summary

This notebook demonstrates a powerful framework for processing MOSDEX optimization model data files using Python and SQLAlchemy. Key achievements include:

1. **Automated Schema Translation**: Converts MOSDEX JSON schemas to SQLAlchemy ORM classes
2. **Data Loading**: Handles both static (INSTANCE) and derived (QUERY) data
3. **Relational Modeling**: Maintains referential integrity through foreign keys
4. **Extensibility**: Works with any MOSDEX-compliant file
5. **Production Ready**: Can be extended to production-scale applications

13.2 Applications

This framework enables:

- **Model Development**: Rapid prototyping of optimization models
- **Data Validation**: Ensuring data integrity before solver submission
- **Results Analysis**: Storing and querying solution data
- **Multi-Model Comparison**: Standardized format for model variants
- **Integration**: Bridge between data sources and optimization solvers

13.3 Future Enhancements

Potential improvements:

1. **Async Processing**: Use asyncio for parallel table creation
2. **Caching**: Cache parsed schemas for repeated use
3. **Visualization**: Generate ER diagrams from MOSDEX
4. **Migration Support**: Use Alembic for schema versioning
5. **Cloud Integration**: Support for cloud databases (AWS RDS, Azure SQL)
6. **Solver Integration**: Direct connection to CPLEX, Gurobi, etc.

13.4 Learning Resources

MOSDEX:

- Official specification: GitHub repository [14]
- Research paper: "MOSDEX: A New Standard for Data Exchange" [1] [2]

SQLAlchemy:

- Official documentation: https://docs.sqlalchemy.org/ [7]
- Tutorial: "Getting Started with SQLAlchemy ORM" [7] [8] [9]

JSON Schema:

- Specification: https://json-schema.org/
- Python implementation: python-jsonschema [5] [6] [10]

References

[4] File content from sailco.ipynb
[3] File content from sailco_2-1.json
[11] File content from MOSDEXSchemaV2-1.json
[1] "MOSDEX: A New Standard for Data Exchange with Optimization Solvers" (optimization-online.org)
[7] "Getting Started with SQLAlchemy ORM for Python" (betterstack.com)
[5] "jsonschema.validators" documentation (python-jsonschema.readthedocs.io)
[2] "MOSDEX: A New Standard for Data Exchange with Optimization Solvers" PDF (optimization-online.org)
[8] "SQLAlchemy Tutorial in Python" (geeksforgeeks.org)
[6] "How to Use JSON Schema to Validate JSON Documents in Python" (builtin.com)
[9] "Setting up a standalone SQLAlchemy 2.0 ORM application" (dev.to)
[14] JeremyBloom/MOSDEX-Reference-Architecture GitHub repository
[13] "SQLAlchemy ORM Tutorial for Python Developers" (auth0.com)
[10] "Schema Validation" (python-jsonschema.readthedocs.io)

[15] [16] [17] [18] [19] [20] [21] [22] [23]

⁂

1. https://optimization-online.org/2023/06/mosdex-a-new-standard-for-data-exchange-with-optimization-solvers/
2. https://optimization-online.org/wp-content/uploads/2023/04/Proposal-for-MOSDEX-Final-Revised-2.pdf
3. sailco_2-1.json
4. sailco.ipynb
5. https://python-jsonschema.readthedocs.io/en/latest/api/jsonschema/validators/
6. https://builtin.com/software-engineering-perspectives/python-json-schema
7. https://betterstack.com/community/guides/scaling-python/sqlalchemy-orm/
8. https://www.geeksforgeeks.org/python/sqlalchemy-tutorial-in-python/
9. https://dev.to/devsnorte/setting-up-a-standalone-sqlalchemy-20-orm-application-298c

10. https://python-jsonschema.readthedocs.io/en/latest/validate/
11. MOSDEXSchemaV2-1.json
12. https://stackoverflow.com/questions/76072801/python-jsonschema-get-the-reason-of-validation-failure
13. https://auth0.com/blog/sqlalchemy-orm-tutorial-for-python-developers/
14. https://github.com/JeremyBloom/MOSDEX-Reference-Architecture
15. https://finance.yahoo.com/news/revolutionizing-crypto-trading-mosdexs-ai-130000418.html
16. https://blog.stackademic.com/mastering-json-schema-validation-with-python-a-developers-guide-0bbf25513630
17. https://easychair.org/smart-program/IOS2022/2022-03-15.html
18. https://www.youtube.com/watch?v=XWtj4zLl_tg
19. https://pypi.org/project/jsonschema-rs/
20. https://www.scribd.com/document/785837327/Program-Book-10-18
21. https://www.datacamp.com/tutorial/sqlalchemy-tutorial-examples
22. https://ddex.net/implementation/
23. https://readthedocs.org/projects/python-jsonschema/downloads/pdf/latest/