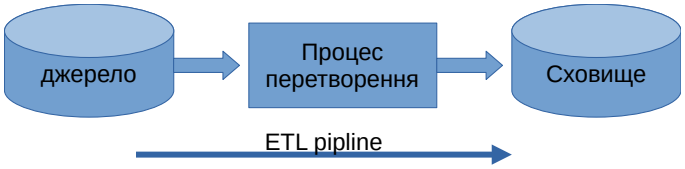
ЛАБОРАТОРНА РОБОТА 4

Тема: Реалізація ETL процесу конвеєру даних

Конвеєр даних представляється низкою завдань: перетворення, фільтри, агрегації та об’єднання кількох джерел, перед виведенням оброблених даних у певний цільовий об’єкт. Через низку кроків та процесів необроблений матеріал форматується та упаковується у вихідне місце, яке найчастіше використовується для зберігання. З точки зору бізнесу, рушійною силою для створення конвеєрів даних є розробка системи для перетворення їхнього найціннішого активу – необроблених даних – на корисну інформацію. Як правило, архітектури конвеєрів даних мають односпрямований потік зв'язку між потоком джерел вхідних даних та системами зберігання вихідних даних:



Конвеєр даних має властивості масштабованості. На його архітектуру впливає як тип даних, які необхідно збирати, так і методології, що використовуються для аналізу даних, для створення сталого середовища даних. Надійний конвеєр даних повинен мати чітко визначені очікування щодо даних, які він обробляє, та результатів, які він має створити. Це включає визначення типів та джерел даних, а також бажаного формату виводу та будь-яких необхідних перетворень або агрегацій. Чітко визначені очікування допомагають забезпечити постійне отримання точних та надійних результатів конвеєром.

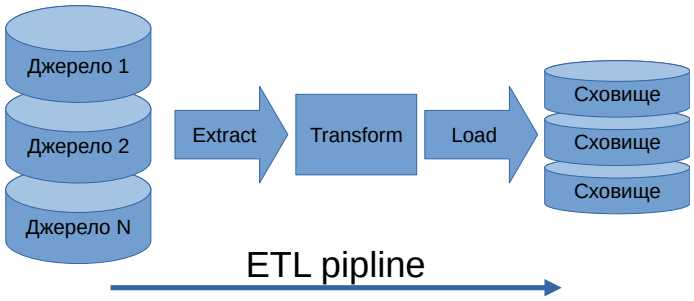
Окрім чітко визначених очікувань, надійний конвеєр даних повинен мати масштабовану архітектуру, яка може обробляти зростаючі обсяги даних без погіршення продуктивності. Це може включати використання розподілених систем або впровадження ефективних алгоритмів для швидкої обробки даних. Відтворюваність та чіткість також є важливими атрибутами надійного конвеєра даних. Конвеєр повинен бути здатним видавати однакові результати щоразу, коли він запускається, а кроки, пов'язані з обробкою даних, повинні бути задокументованими та легкими для розуміння. Це допомагає забезпечити легке обслуговування та зміну конвеєра за потреби.

Основною підготовкою є розуміння даних, що впливає на визначення контексту і потік загального конвеєра даних. При проєктуванні та реалізації конвеєра важливо мати чітке розуміння як вхідних, так і вихідних структур даних. Це включає знання структур даних, будь-яких потенційних проблем з даними, наприклад, таких як пошкодження та частоту створення нових вхідних даних. Для вихідних даних важливо розуміти структурні вимоги, щоб забезпечити послідовне отримання бажаного результату від конвеєра. Також важливо переконатися, що конвеєр служить певній меті та допомагає досягти цілей проекту. Якщо конвеєр не дає бажаного результату або не служить певній меті, його можна вважати непотрібним, а витрачені на його створення ресурси - втраченими.

При проєктуванні та розробці конвеєру даних необхідно розглянуті типи з'єднань, необхідні для безпечного зв'язку джерела (джерел) вхідних даних із середовищами розробки та виробництва, а також метод підключення до місця виведення. Також потрібно визначити підхід до конвеєра та рівень обробки помилок, необхідний для забезпечення довгострокової стабільності та надійності конвеєра.

Один із типів конвеєру даних ґрунтується на реалізації популярного процесу ETL (Extract, Transform, and Load) - «Видобування, Трансформація та Завантаження», дані спочатку витягуються з джерела, потім трансформуються та форматуються певним чином, і, нарешті, завантажуються в кінцеве місце зберігання. Ці конвеєри корисні для організації та підготовки

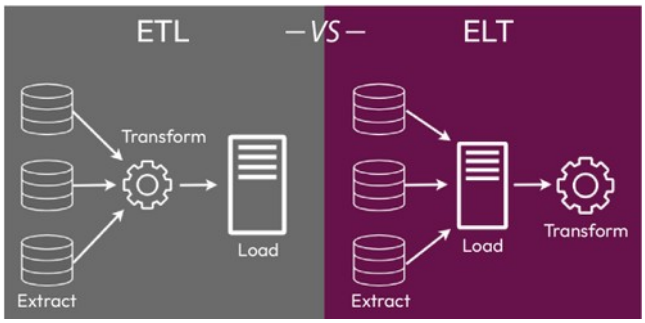
даних для майбутніх цілей, таких як безперебійне та ефективне виконання аналізу та створення моделей:



Ще одним популярним процесом, що застосовується для конвеєрів даних, є ELT - «Видобування, Завантаження та Трансформація» (Extract, Load, and Transform), який відрізняється від ETL перетворенням даних вже після їх завантаження в сховище. Вибір методу залежить від конкретних вимог та характеристик задіяних систем, а також даних, що переміщуються. Ось кілька факторів, які ви можете врахувати, вибираючи між ETL та ELT:

- обсяг даних: якщо обсяг даних дуже великий, ELT може бути ефективнішим, оскільки етап перетворення можна виконувати паралельно в цільовій системі;
- вимоги до перетворення даних: якщо дані потребують складних перетворень, може бути легше виконати перетворення в цільовій системі за допомогою ELT;
- можливості вихідної системи: якщо вихідна система не в змозі виконати необхідні перетворення, ETL може бути єдиним варіантом;
- можливості цільової системи: якщо цільова система не в змозі ефективно обробляти фазу завантаження процесу ETL, ELT може бути кращим варіантом;
- затримка даних: якщо потрібне переміщення даних у режимі реального часу, ELT може бути кращим вибором, оскільки він дозволяє швидше завантажувати та перетворювати дані.

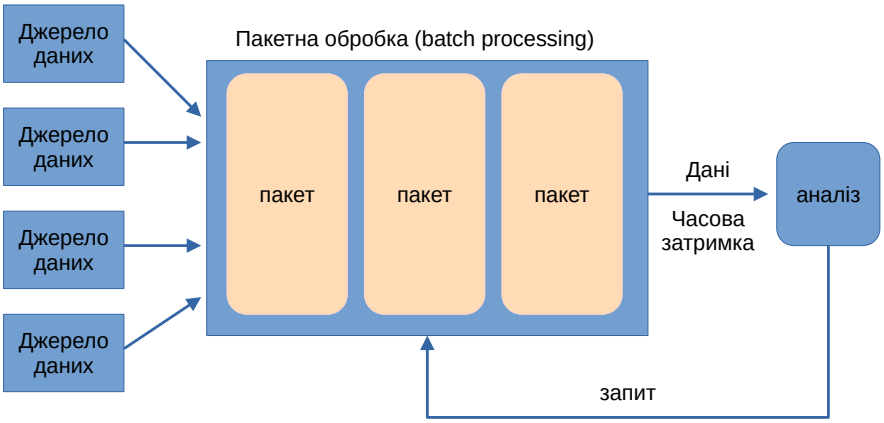
Загалом, ETL частіше використовується там, де вихідна та цільова системи різні, і дані потрібно перетворювати певним чином, перш ніж їх завантажувати в цільову систему. ELT частіше використовується тоді, коли цільова система є найпотужнішою в загальній системі і може самостійно виконати етап перетворення:



Існує три основні типи ETL-конвеєрів: з пакетною обробкою, з потоковою передачею та хмарна обробка.

Пакетна обробка – це метод обробки даних, який передбачає розділення великого обсягу даних на менші частини або пакети та обробку кожного пакета окремо. Такий підхід застосовується для проєктів, що вимагають обробки великого обсягу даних, а подальше їх використання вимагає лише асинхронної доступності. Пакетна обробка є популярною для обробки великих обсягів даних, оскільки дозволяє реалізовувати процес, який обробляє пакети по одному через конвеєр, а не обробляє усі дані одночасно. Застосовується при переміщені великих обсягів даних через регулярні проміжки часу.

Одним із прикладів системи пакетної обробки є компанія, яка обробляє замовлення клієнтів для інтернет-магазину. Компанія щодня отримує великий обсяг замовлень і повинна обробляти ці дані для оновлення рівня запасів, створення рахунків-фактур та виконання замовлень. Для цього вони використовують систему пакетної обробки, щоб розбити дані на менші фрагменти та пропустити кожен фрагмент через конвеєр даних. Це дозволяє ефективно обробляти дані, не перевантажувати системи та не спричиняти затримок в обробці замовлень:



Метод потокової передачі — це рішення для обробки даних у реальному часі і необхідне, коли проект потребує негайної обробки нових даних. Метод потокової передачі дозволяє даним надходити безперервно і може бути змінним та піддаватися раптовим змінам структури. Для вирішення цих проблем зазвичай використовується комбінація конвеєрів даних. Існує кілька передових інструментів, таких як Apache Storm та Apache Samza, які можуть ефективно обробляти дані в реальному часі. Прикладом використання методу потокової передачі є веб-сайт електронної комерції, який потребує обробки даних користувачів у реальному часі під час здійснення покупок. Використання обробки даних у реальному часі в поєднанні зі штучним інтелектом/модельним навчанням може покращити досвід покупок для користувачів.

Пакетна обробка використовується для великих обсягів даних, які повинні бути доступні лише асинхронно, потокова передача використовується для обробки даних у режимі реального часу, а хмарна обробка використовується для обробки даних у середовищах хмарних обчислень.

Існує кілька ключових переваг автоматизації конвеєрів ETL:

- автоматизація процесу ETL може полегшити доступ до даних та їх використання широким колом користувачів, оскільки процес витягу, перетворення та завантаження даних спрощується та стає ефективнішим;
- надійна доступність даних — конвеєри розроблені для регулярної роботи та їх можна налаштувати для обробки будь-яких змін або оновлень вихідних даних;
- орієнтація на команду, оскільки автоматизація конвеєрів ETL може звільнити членів команди, щоб вони могли зосередитися на важливіших завданнях, таких як аналіз даних, розробка моделей машинного навчання, тощо;
- спрощується та пришвидшується адаптація нових членів команди, оскільки процес вилучення та підготовки даних вже спрощений та автоматизований, що зменшує потребу нових співробітників у вивченні складних ручних процесів;
- допомагає в управлінні схемами, забезпечує перетворення даних належним чином та завантажує в цільову систему.

Використання конвеєрів ETL в організаціях надають наступні переваги:

- дозволяють розробникам та інженерам зосередитися на корисних завданнях і не турбуватись додатково про дані;
- звільняють час для розробників, інженерів та науковців для зосередження на фактичній роботі;
- допомагають організаціям ефективно та систематично переміщувати дані з одного місця в інше та перетворювати їх у потрібний формат;

- реалізація міграції даних зі застарілої платформи до хмари та навпаки;
- централізація джерел даних для отримання їх консолідованого представлення;
- забезпечення стабільних джерел даних для програм, керованих даними, та інструментів аналізу даних;
- виконання ролі плану для організаційних даних, слугує єдиним джерелом достовірної інформації.

До загальних переваг конвеєрів ETL слід віднести економію коштів у довгостроковій перспективі та допомогу в розширенні бізнесу.

Практична частина

Розглянемо класичний бізнес-сценарій для конвеєра даних (Data Pipeline). Припустимо, що деякій компанії необхідно:

- щоденно завантажувати котирування активів;
- розраховувати дохідність та технічні індикатори;
- зберігати агреговані дані у внутрішньому сховищі.

У наведеному нижче прикладі реалізовано наступну схему:
EXTRACT ⇒ TRANSFORM ⇒ DATA QUALITY CHECK ⇒ LOAD ⇒ Аналітика

Це повноцінний ETL-pipeline для побудови невеликого фінансового сховища даних (Financial Data Warehouse). На прикладі Yahoo Finance код призначено для автоматизованого збору, обробки та аналізу біржових даних.

1. Автоматичний збір фінансових даних (Data Ingestion)
Для отримання даних фондового ринку із зовнішнього джерела реалізовано клас DataExtractor. Використовується бібліотека yfinance і для прикладу завантажуються дані для тікерів: AAPL, GOOGL, MSFT, TSLA, AMZN. У результаті формується DataFrame із часовими рядами біржових котирувань за даними: Open, High, Low, Close, Volume.
Для логування роботи ETL-процесу використовується self.logger, який створюється на основі стандартного пакету Python logging. При утворені логеру задається певне ім’я і у подальшому через логер робляться записи інформації про виконання програми, попередження та помилки. Тому логер є важливим елементом моніторингу та діагностики ETL-pipeline.

```
2026-03-05 23:24:20,609 - __main__ - INFO - Конфігурація завантажена: ['AAPL', 'GOOGL', 'MSFT', 'TSLA', 'AMZN']
2026-03-06 00:01:05,508 - __main__ - INFO - Конфігурація завантажена: ['AAPL', 'GOOGL', 'MSFT', 'TSLA', 'AMZN']
2026-03-06 00:01:05,511 - ETLOrchestrator - INFO - ============================================================
2026-03-06 00:01:05,512 - ETLOrchestrator - INFO - ПОЧАТОК ЩОДЕННОГО ETL-ПРОЦЕСУ
2026-03-06 00:01:05,512 - ETLOrchestrator - INFO - ============================================================
2026-03-06 00:01:05,513 - ETLOrchestrator - INFO -
--- ЕТАП 1: ВИТЯГ ДАНИХ ---
2026-03-06 00:01:05,513 - DataExtractor - INFO - Завантаження даних для AAPL
2026-03-06 00:01:06,691 - DataExtractor - INFO - Завантажено 251 днів для AAPL
2026-03-06 00:01:06,692 - DataExtractor - INFO - Завантаження даних для GOOGL
2026-03-06 00:01:06,849 - DataExtractor - INFO - Завантажено 251 днів для GOOGL
2026-03-06 00:01:06,850 - DataExtractor - INFO - Завантаження даних для MSFT
```

Основним методом отримання даних для одного фінансового інструмента є extract_ticker(), через який і формується згадуваний вище DataFrame. Для контролю, щодо надходження даних, перевіряється властивість empty створеного DataFrame. Якщо дані не надійшли, то буде відповідне попередження. До отриманих даних також додаються метадані — окремі колонки з тікером, датою витягу та джерелом даних. Наявність тікеру у подальшому дозволить об’єднувати дані різних акцій, дата витягу дозволяє контролювати актуальність та проводити аудит даних, а фіксація джерела даних дозволяє значно розширити операції та аналітику даних і тому є типовою практикою у Data Warehouse.

Кінцевий результат формується у вигляді «результат - словник» і є зручною структурою для етапу трансформації, на якому дані очищуються, обчислюються метрики та створюються індикатори.

Етап трансформації даних поділяється на декілька складових, частина з яких є типовою для будь-яких даних — видалення дублікатів, заповнення пропущених значень, а інші застосовуються виключно для фінансових даних акцій компаній на біржі — різні типи дохідності, біржові технічні індикатори.

2. Очищення та підготовка даних (Data Cleaning)

Основною задачею на цьому етапі є підготовка даних до подальшої аналітики. Для цього призначено метод clean_data(). Дублікати можуть вплинути на висновки бізнес-аналізу, а їх поява інколи обумовлена особливостями API, що забезпечує створення та доступ до даних. Для видалення дублікатів використовується df_clean = df_clean[~df_clean.index.duplicated()].

Ще однією проблемою є пропущені значення — замість певного значення наявне позначення типу NaN (*Not-a-Number*, «не число») і подальша їх обробка просто неможлива, що впливає на аналіз даних часових фінансових рядів. Тому для зменшення впливу невизначеності або застосовується повторення попереднього значення, або середнє між попереднім та наступним. Для цього використовується: df_clean[col] = df_clean[col].fillna(method='ffill'), результатом якого є повторення попереднього значення.

Наступні складові етапу трансформації, а саме: обчислення фінансових показників, розрахунок технічних індикаторів, додавання часових вимірів, за своєю суттю збагачують наявні. Під збагаченням даних розуміють процес додавання нових атрибутів або показників до існуючих даних, щоб зробити їх більш інформативними, більш корисними для аналізу, моделей та звітів.

3. Обчислення фінансових показників

На цьому етапі основною задачею є перетворення сирих біржових даних на певні аналітичні метрики: просту, логарифмічну та накопичену дохідності. Обчислення виконуються за допомогою методу calculate_returns().

4. Розрахунок технічних індикаторів

На цьому етапі виконується підготовка даних для алгоритмічної торгівлі та фінансової аналітики, що реалізується через метод calculate_technical_indicators(). Розраховуються: ковзна середня (SMA — Simple Moving Average) за для визначення трендів ринку, експоненціальна ковзна середня (EMA — Exponential Moving Average), яка є кращою для короткострокової торгівлі та швидкого виявлення розвороту ціни, волатильність для оцінки рівня ризику активу, RSI (Relative Strength Index), що дозволяє визначити специфічні характеристики «перекупленість» та «перепроданість» активу, а також сходження/розбіжність ковзних середніх MACD (Moving Average Convergence/Divergence), який використовується для оцінки ринку, напряму тренду та імпульсу ринку.

5. Додавання календарних ознак

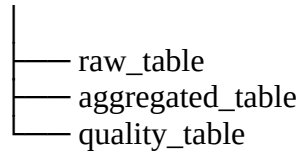
Первинні дані є часовим рядом, для яких можна точно визначити такі календарні значення, як день тижня, місяць, квартал та рік. Це в подальшому дозволить виконати сезонний аналіз, машинне навчання та аналітику. Визначення та додавання таких календарних параметрів є типовою практикою для Data Warehouse. У наведеному нижче коді для вирішення цієї проблеми використовується метод aggregate_daily().

Також на етапі фінального формування до даних додаються такі метадані, як тікери та дата виконання розрахунків, що дозволяє відстежувати джерело даних та вести аудит ETL-pipeline.

Таким чином вихідний датасет буде містити наступні колонки: Open (ціна відкриття), High (максимум), Low(мінімум), Close(ціна закриття), Volume(обсяг), simple_return(проста доходність), log_return (лог-доходність), cumulative_return (доходність), sma_20 (ковзна середня), ema_20(експоненційна середня), volatility (волатильність), rsi(RSI), macd(MACD),

macd_signal(сигнал MACD), day_of_week(день тижня), month(місяць), year(рік), quarter(квартал), ticker(тикер), calculation_date(дата обчислення). Створений аналітичний фінансовий датасет можна використати для фінансової аналітики, визначення трейдингових стратегій, машинного навчання, ризик-аналізу, портфельної оптимізації та BI-звітів.

6. Завантаження даних у сховище (Load)
На цьому етапі реалізується завантаження підготовлених даних у базу даних SQLite, як найпростіший варіант сховища даних. Клас DataLoader реалізує повний цикл роботи з БД: методи підключення до бази даних та закриття з’єднання, метод безпечної обробки значень перед записом, що перетворює їх з pandas у звичайні Python-типи, створює таблиці у БД: сирих даних, агрегованих даних, метрик якості даних (повнота, точність, узгодженість, актуальність, валідність, унікальність, загальний рейтинг, рекомендації). Завантаження сирих даних здійснюється через метод load_raw_data(), завантаження агрегованих даних — через метод load_aggregated_data(), завантаження метрик якості — через метод load_quality_metrics(). На цьому етапі у коді активно використовуються логери: self.logger.info() та self.logger.error(), що допомагає відстежувати роботу ETL та знаходити помилки. У результаті в БД SQLite з’являється структуроване сховище даних:

database.db


що дозволяє тепер виконувати SQL-запити, будувати звіти, підключати BI-інструменти та проводити фінансову аналітику.

7. Центральний координатор (оркестратор)
Роль центрального координатора (керуючого модуля) ETL-процесу виконує ETLOrchestrator. Основним його завданням є послідовно запускати всі етапи ETL-pipeline та контролювати їх виконання. Таким чином, він керує всім процесом обробки даних від витягу до збереження та перевірки якості та об’єднує всі частини pipeline.
Оркестратор створює об'єкти трьох основних етапів ETL: DataExtractor - отримує дані з джерела, DataTransformer - очищує та обчислює показники, DataLoader, зберігає дані у базі.

Приклад реалізації ETL

```
import yfinance as yf
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import sqlite3
import logging
from typing import Dict, List, Optional
import json

# Налаштування логування
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('etl_pipeline.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

# ==========================================
# КОНФІГУРАЦІЯ
# ==========================================

class ETLConfig:
    """Конфігурація ETL-процесу"""
```

```
    def __init__(self, config_file=None):
        # Типові налаштування
        self.tickers = ['AAPL', 'GOOGL', 'MSFT', 'TSLA', 'AMZN']
        self.lookback_days = 365  # Завантажуємо останній рік
        self.update_frequency = 'daily'  # daily, weekly, monthly
        self.calculation_window = 20  # Для ковзних середніх
        self.rsi_period = 14
        self.db_path = 'financial_data.db'
        self.raw_table = 'raw_quotes'
        self.aggregated_table = 'aggregated_metrics'
        self.quality_table = 'data_quality'

        # Завантаження з файлу, якщо є
        if config_file:
            with open(config_file, 'r') as f:
                config_data = json.load(f)
                for key, value in config_data.items():
                    setattr(self, key, value)

        logger.info(f"Конфігурація завантажена: {self.tickers}")

# ============================================
# ЕТАП 1: EXTRACT (ВИТЯГ)
# ============================================

class DataExtractor:
    """
    Витяг даних з Yahoo Finance
    """

    def __init__(self, config: ETLConfig):
        self.config = config
        self.logger = logging.getLogger('DataExtractor')

    def extract_ticker(self, ticker: str, start_date: datetime, end_date: datetime) -> pd.DataFrame:
        """
        Витяг даних для одного тікера
        """
        try:
            self.logger.info(f"Завантаження даних для {ticker}")

            # Завантаження з Yahoo Finance
            df = yf.download(
                ticker,
                start=start_date,
                end=end_date,
                progress=False,
                auto_adjust=True
            )

            if df.empty:
                self.logger.warning(f"Немає даних для {ticker}")
                return None

            # Додаємо метадані
            df['ticker'] = ticker
            df['extraction_date'] = datetime.now()
            df['data_source'] = 'Yahoo Finance'

            self.logger.info(f"Завантажено {len(df)} днів для {ticker}")
            return df

        except Exception as e:
            self.logger.error(f"Помилка завантаження {ticker}: {e}")
            return None

    def extract_all(self) -> Dict[str, pd.DataFrame]:
        """
        Витяг даних для всіх тікерів
        """
        end_date = datetime.now()
        start_date = end_date - timedelta(days=self.config.lookback_days)

        data = {}
        for ticker in self.config.tickers:
            df = self.extract_ticker(ticker, start_date, end_date)
            if df is not None:
                data[ticker] = df

        self.logger.info(f"Витяг завершено: {len(data)} тікерів")
        return data

# ============================================
# ЕТАП 2: TRANSFORM (ТРАНСФОРМАЦІЯ)
# ============================================

class DataTransformer:
    """
    Трансформація даних: очищення, розрахунок метрик, агрегація
    """
```

```
    def __init__(self, config: ETLConfig):
        self.config = config
        self.logger = logging.getLogger('DataTransformer')

    def clean_data(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Очищення даних
        """
        df_clean = df.copy()

        # Видаляємо дублікати
        before = len(df_clean)
        df_clean = df_clean[~df_clean.index.duplicated(keep='first')]
        after = len(df_clean)

        if before > after:
            self.logger.info(f"Видалено {before - after} дублікатів")

        # Заповнюємо пропуски
        if df_clean.isnull().any().any():
            # Forward fill для цін
            price_cols = ['Open', 'High', 'Low', 'Close', 'Volume']
            for col in price_cols:
                if col in df_clean.columns:
                    df_clean[col] = df_clean[col].fillna(method='ffill')

            self.logger.info(f"Заповнено пропуски")

        return df_clean

    def calculate_returns(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Розрахунок доходностей
        """
        df_returns = df.copy()

        # Проста доходність
        df_returns['simple_return'] = df_returns['Close'].pct_change()

        # Логарифмічна доходність
        df_returns['log_return'] = np.log(df_returns['Close'] / df_returns['Close'].shift(1))

        # Накопичена доходність
        df_returns['cumulative_return'] = (1 + df_returns['simple_return']).cumprod() - 1

        self.logger.info("Розраховано доходності")
        return df_returns

    def calculate_technical_indicators(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Розрахунок технічних індикаторів
        """
        df_indicators = df.copy()
        window = self.config.calculation_window
        rsi_period = self.config.rsi_period

        # Ковзні середні
        df_indicators['sma_20'] = df_indicators['Close'].rolling(window=window).mean()
        df_indicators['ema_20'] = df_indicators['Close'].ewm(span=window, adjust=False).mean()

        # Волатильність
        df_indicators['volatility'] = df_indicators['simple_return'].rolling(window=window).std() *
np.sqrt(252)

        # RSI
        delta = df_indicators['Close'].diff()
        gain = delta.where(delta > 0, 0).rolling(window=rsi_period).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=rsi_period).mean()
        rs = gain / loss
        df_indicators['rsi'] = 100 - (100 / (1 + rs))

        # MACD
        exp1 = df_indicators['Close'].ewm(span=12, adjust=False).mean()
        exp2 = df_indicators['Close'].ewm(span=26, adjust=False).mean()
        df_indicators['macd'] = exp1 - exp2
        df_indicators['macd_signal'] = df_indicators['macd'].ewm(span=9, adjust=False).mean()

        self.logger.info("Розраховано технічні індикатори")
        return df_indicators

    def aggregate_daily(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Денна агрегація (для щоденного звіту)
        """
        # Групуємо по датах (вже є індексом)
        df_agg = df.copy()

        # Додаємо календарні ознаки
        df_agg['day_of_week'] = df_agg.index.dayofweek
        df_agg['month'] = df_agg.index.month
        df_agg['year'] = df_agg.index.year
```

```
        df_agg['quarter'] = df_agg.index.quarter

        return df_agg

    def transform_ticker(self, df: pd.DataFrame, ticker: str) -> pd.DataFrame:
        """
        Повний цикл трансформації для одного тікера
        """
        self.logger.info(f"Трансформація даних для {ticker}")

        # Очищення
        df_clean = self.clean_data(df)

        # Розрахунок доходностей
        df_returns = self.calculate_returns(df_clean)

        # Розрахунок індикаторів
        df_indicators = self.calculate_technical_indicators(df_returns)

        # Агрегація
        df_final = self.aggregate_daily(df_indicators)

        # Додаємо метадані
        df_final['ticker'] = ticker
        df_final['calculation_date'] = datetime.now()

        self.logger.info(f"Трансформацію завершено: {len(df_final)} записів")
        return df_final

    def transform_all(self, raw_data: Dict[str, pd.DataFrame]) -> Dict[str, pd.DataFrame]:
        """
        Трансформація всіх даних
        """
        transformed = {}
        for ticker, df in raw_data.items():
            transformed[ticker] = self.transform_ticker(df, ticker)

        self.logger.info(f"Трансформацію всіх даних завершено")
        return transformed

# ============================================
# ЕТАП 3: LOAD (ЗАВАНТАЖЕННЯ)
# ============================================

class DataLoader:
    """
    Фінальна версія з правильною обробкою Timestamp
    """

    def __init__(self, config: ETLConfig):
        self.config = config
        self.logger = logging.getLogger('DataLoader')
        self.connection = None

    def connect(self):
        """Підключення до бази даних"""
        try:
            self.connection = sqlite3.connect(self.config.db_path)
            self.logger.info(f"Підключено до БД: {self.config.db_path}")
            return True
        except Exception as e:
            self.logger.error(f"Помилка підключення: {e}")
            return False

    def close(self):
        """Закриття з'єднання"""
        if self.connection:
            self.connection.close()
            self.logger.info("З'єднання закрито")

    def _get_value(self, value, default=None):
        """
        Покращений метод для безпечного отримання скалярного значення
        """
        if value is None:
            return default

        # ВИПРАВЛЕННЯ: Обробка Timestamp
        if isinstance(value, pd.Timestamp):
            return value.to_pydatetime()  # Конвертуємо в datetime

        # Якщо це Series, беремо перше значення
        if isinstance(value, pd.Series):
            if len(value) > 0:
                val = value.iloc[0]
                # Рекурсивно обробляємо отримане значення
                return self._get_value(val, default)
            return default

        # Якщо це numpy тип, конвертуємо в Python тип
        if hasattr(value, 'item'):
```

```
            try:
                return value.item()
            except:
                pass

        # Якщо це вже скаляр
        if pd.isna(value):
            return default

        return value

    def create_tables(self):
        """Створення таблиць, якщо їх немає"""
        cursor = self.connection.cursor()

        # Таблиця для сирих даних
        cursor.execute(f'''
            CREATE TABLE IF NOT EXISTS {self.config.raw_table} (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                ticker TEXT NOT NULL,
                date DATE NOT NULL,
                open REAL,
                high REAL,
                low REAL,
                close REAL,
                volume INTEGER,
                extraction_date TIMESTAMP,
                data_source TEXT,
                UNIQUE(ticker, date)
            )
        ''')

        # Таблиця для агрегованих метрик
        cursor.execute(f'''
            CREATE TABLE IF NOT EXISTS {self.config.aggregated_table} (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                ticker TEXT NOT NULL,
                date DATE NOT NULL,
                open REAL,
                high REAL,
                low REAL,
                close REAL,
                volume INTEGER,
                simple_return REAL,
                log_return REAL,
                cumulative_return REAL,
                sma_20 REAL,
                ema_20 REAL,
                volatility REAL,
                rsi REAL,
                macd REAL,
                macd_signal REAL,
                day_of_week INTEGER,
                month INTEGER,
                year INTEGER,
                quarter INTEGER,
                calculation_date TIMESTAMP,
                UNIQUE(ticker, date)
            )
        ''')

        # Таблиця для метрик якості
        cursor.execute(f'''
            CREATE TABLE IF NOT EXISTS {self.config.quality_table} (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                ticker TEXT NOT NULL,
                calculation_date TIMESTAMP,
                completeness REAL,
                accuracy REAL,
                consistency REAL,
                timeliness REAL,
                validity REAL,
                uniqueness REAL,
                overall REAL,
                recommendation TEXT
            )
        ''')

        self.connection.commit()
        self.logger.info("Таблиці створено/перевірено")

    def load_raw_data(self, data: Dict[str, pd.DataFrame]):
        """
        Покращене завантаження сирих даних
        """
        cursor = self.connection.cursor()
        total_inserted = 0
        errors = 0

        for ticker, df in data.items():
            self.logger.info(f"Завантаження сирих даних для {ticker}")
```

```
            for idx, row in df.iterrows():
                try:
                    # ВИПРАВЛЕННЯ: конвертуємо Timestamp в datetime
                    date_val = self._get_value(idx)
                    if isinstance(date_val, pd.Timestamp):
                        date_val = date_val.to_pydatetime().date()

                    # Безпечне отримання значень
                    open_val = self._get_value(row.get('Open', None))
                    high_val = self._get_value(row.get('High', None))
                    low_val = self._get_value(row.get('Low', None))
                    close_val = self._get_value(row.get('Close', None))
                    volume_val = self._get_value(row.get('Volume', None))
                    extraction_val = self._get_value(row.get('extraction_date', datetime.now()))
                    source_val = self._get_value(row.get('data_source', 'Yahoo Finance'))

                    cursor.execute(f'''
                        INSERT OR REPLACE INTO {self.config.raw_table}
                        (ticker, date, open, high, low, close, volume, extraction_date, data_source)
                        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
                    ''', (
                        ticker,
                        date_val,
                        open_val,
                        high_val,
                        low_val,
                        close_val,
                        volume_val,
                        extraction_val,
                        source_val
                    ))
                    total_inserted += 1

                except Exception as e:
                    errors += 1
                    if errors <= 5:
                        self.logger.error(f"Помилка вставки {ticker} {idx}: {e}")

        self.connection.commit()
        self.logger.info(f"Завантажено {total_inserted} сирих записів (помилок: {errors})")

    def load_aggregated_data(self, data: Dict[str, pd.DataFrame]):
        """
        Завантаження агрегованих даних
        """
        cursor = self.connection.cursor()
        total_inserted = 0
        errors = 0

        for ticker, df in data.items():
            self.logger.info(f"Завантаження агрегованих даних для {ticker}")

            for idx, row in df.iterrows():
                try:
                    # ВИПРАВЛЕННЯ: конвертуємо Timestamp
                    date_val = self._get_value(idx)
                    if isinstance(date_val, pd.Timestamp):
                        date_val = date_val.to_pydatetime().date()

                    values = (
                        ticker,
                        date_val,
                        self._get_value(row.get('Open', None)),
                        self._get_value(row.get('High', None)),
                        self._get_value(row.get('Low', None)),
                        self._get_value(row.get('Close', None)),
                        self._get_value(row.get('Volume', None)),
                        self._get_value(row.get('simple_return', None)),
                        self._get_value(row.get('log_return', None)),
                        self._get_value(row.get('cumulative_return', None)),
                        self._get_value(row.get('sma_20', None)),
                        self._get_value(row.get('ema_20', None)),
                        self._get_value(row.get('volatility', None)),
                        self._get_value(row.get('rsi', None)),
                        self._get_value(row.get('macd', None)),
                        self._get_value(row.get('macd_signal', None)),
                        self._get_value(row.get('day_of_week', None)),
                        self._get_value(row.get('month', None)),
                        self._get_value(row.get('year', None)),
                        self._get_value(row.get('quarter', None)),
                        datetime.now()
                    )

                    cursor.execute(f'''
                        INSERT OR REPLACE INTO {self.config.aggregated_table}
                        (ticker, date, open, high, low, close, volume,
                         simple_return, log_return, cumulative_return,
                         sma_20, ema_20, volatility, rsi, macd, macd_signal,
                         day_of_week, month, year, quarter, calculation_date)
                        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

```
                    ''', values)

                    total_inserted += 1

                except Exception as e:
                    errors += 1
                    if errors <= 5:
                        self.logger.error(f"Помилка вставки агрегованих {ticker} {idx}: {e}")

        self.connection.commit()
        self.logger.info(f"Завантажено {total_inserted} агрегованих записів (помилок: {errors})")

    def load_quality_metrics(self, quality_report: pd.DataFrame):
        """
        Завантаження метрик якості
        """
        cursor = self.connection.cursor()
        total_inserted = 0

        for ticker, row in quality_report.iterrows():
            try:
                cursor.execute(f'''
                    INSERT INTO {self.config.quality_table}
                    (ticker, calculation_date, completeness, accuracy, consistency,
                     timeliness, validity, uniqueness, overall, recommendation)
                    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                ''', (
                    ticker,
                    datetime.now(),
                    self._get_value(row.get('Completeness', None)),
                    self._get_value(row.get('Accuracy', None)),
                    self._get_value(row.get('Consistency', None)),
                    self._get_value(row.get('Timeliness', None)),
                    self._get_value(row.get('Validity', None)),
                    self._get_value(row.get('Uniqueness', None)),
                    self._get_value(row.get('Overall', None)),
                    self._get_value(row.get('Recommendation', None))
                ))
                total_inserted += 1

            except Exception as e:
                self.logger.error(f"Помилка вставки метрик для {ticker}: {e}")

        self.connection.commit()
        self.logger.info(f"Завантажено {total_inserted} записів метрик якості")

# ============================================
# ГОЛОВНИЙ ETL-ОРКЕСТРАТОР
# ============================================

class ETLOrchestrator:
    """
    Виправлений оркестратор ETL-процесу
    """

    def __init__(self, config: ETLConfig):
        self.config = config
        self.logger = logging.getLogger('ETLOrchestrator')

        self.extractor = DataExtractor(config)
        self.transformer = DataTransformer(config)
        self.loader = DataLoader(config)

    def run_daily_etl(self) -> bool:
        """
        Запуск щоденного ETL-процесу
        """
        self.logger.info("="*60)
        self.logger.info("ПОЧАТОК ЩОДЕННОГО ETL-ПРОЦЕСУ")
        self.logger.info("="*60)

        try:
            # ЕТАП 1: EXTRACT
            self.logger.info("\n--- ЕТАП 1: ВИТЯГ ДАНИХ ---")
            raw_data = self.extractor.extract_all()

            if not raw_data:
                self.logger.error("Немає даних для обробки")
                return False

            # ЕТАП 2: TRANSFORM
            self.logger.info("\n--- ЕТАП 2: ТРАНСФОРМАЦІЯ ДАНИХ ---")
            transformed_data = self.transformer.transform_all(raw_data)

            # ЕТАП 3: LOAD
            self.logger.info("\n--- ЕТАП 3: ЗАВАНТАЖЕННЯ ДАНИХ ---")

            if self.loader.connect():
                # Створення таблиць
                self.loader.create_tables()
```

```
                # Завантаження даних
                self.loader.load_raw_data(raw_data)
                self.loader.load_aggregated_data(transformed_data)

                # ОЦІНКА ЯКОСТІ
                self.logger.info("\n--- ОЦІНКА ЯКОСТІ ДАНИХ ---")

                # ВИПРАВЛЕННЯ: імпортуємо pandas тут, якщо потрібно
                import pandas as pd
                from data_quality import DataQualityAssessor

                # Використовуємо наш DataQualityAssessor
                assessor = DataQualityAssessor(raw_data)
                quality_report = assessor.comprehensive_report()

                # Виводимо звіт
                print("\n" + "="*60)
                print("ЗВІТ ПРО ЯКІСТЬ ДАНИХ")
                print("="*60)
                print(quality_report)

                # Завантажуємо в БД
                self.loader.load_quality_metrics(quality_report)

                # Закриття з'єднання
                self.loader.close()

            self.logger.info("="*60)
            self.logger.info("ETL-ПРОЦЕС УСПІШНО ЗАВЕРШЕНО")
            self.logger.info("="*60)

            return True

        except Exception as e:
            self.logger.error(f"КРИТИЧНА ПОМИЛКА ETL: {e}")
            import traceback
            traceback.print_exc()
            return False

# ==========================================
# АНАЛІТИЧНІ ЗАПИТИ ДО СХОВИЩА
# ==========================================

class DataWarehouseAnalytics:
    """
    Аналітичні запити до сховища даних
    """

    def __init__(self, db_path: str):
        self.db_path = db_path
        self.connection = None

    def connect(self):
        self.connection = sqlite3.connect(self.db_path)

    def close(self):
        if self.connection:
            self.connection.close()

    def get_daily_report(self, ticker: str, days: int = 30) -> pd.DataFrame:
        """
        Отримання денного звіту для тікера
        """
        query = f"""
            SELECT date, close, simple_return, cumulative_return,
                   sma_20, rsi, volatility
            FROM aggregated_metrics
            WHERE ticker = '{ticker}'
            ORDER BY date DESC
            LIMIT {days}
        """
        return pd.read_sql_query(query, self.connection)

    def get_quality_history(self, ticker: str) -> pd.DataFrame:
        """
        Історія метрик якості для тікера
        """
        query = f"""
            SELECT calculation_date, completeness, accuracy,
                   consistency, timeliness, validity, uniqueness, overall
            FROM data_quality
            WHERE ticker = '{ticker}'
            ORDER BY calculation_date DESC
        """
        return pd.read_sql_query(query, self.connection)

    def get_top_performers(self, date: str, n: int = 5) -> pd.DataFrame:
        """
        Топ-5 тікерів за доходністю на певну дату
        """
        query = f"""
```

```
            SELECT ticker, close, simple_return, cumulative_return
            FROM aggregated_metrics
            WHERE date = '{date}'
            ORDER BY simple_return DESC
            LIMIT {n}
        """
        return pd.read_sql_query(query, self.connection)

    def get_correlation_matrix(self, tickers: List[str]) -> pd.DataFrame:
        """
        Кореляційна матриця для списку тікерів
        """
        returns_data = {}

        for ticker in tickers:
            query = f"""
                SELECT date, simple_return
                FROM aggregated_metrics
                WHERE ticker = '{ticker}'
                ORDER BY date
            """
            df = pd.read_sql_query(query, self.connection)
            if not df.empty:
                df.set_index('date', inplace=True)
                returns_data[ticker] = df['simple_return']

        returns_df = pd.DataFrame(returns_data)
        return returns_df.corr()

# ============================================
# ПРИКЛАД ВИКОРИСТАННЯ
# ============================================

if __name__ == "__main__":
    # Створення конфігурації
    config = ETLConfig()

    # Додавання нашого DataQualityAssessor
    from data_quality import DataQualityAssessor  # Імпортуємо наш попередній клас

    # Запуск ETL
    orchestrator = ETLOrchestrator(config)
    success = orchestrator.run_daily_etl()

    if success:
        # Аналітика
        analytics = DataWarehouseAnalytics(config.db_path)
        analytics.connect()

        # Отримання звіту
        print("\n📊 ДЕННИЙ ЗВІТ ДЛЯ AAPL:")
        report = analytics.get_daily_report('AAPL', days=5)
        print(report)

        # Топ-виконавці
        print("\n🏆 ТОП-5 ТІКЕРІВ:")
        top = analytics.get_top_performers(
            (datetime.now() - timedelta(days=1)).strftime('%Y-%m-%d')
        )
        print(top)

        # Історія якості
        print("\n📈 ІСТОРІЯ ЯКОСТІ ДЛЯ TSLA:")
        quality = analytics.get_quality_history('TSLA')
        print(quality)

        analytics.close()

# ============================================
# ДОДАТКОВО: ПЛАНУВАЛЬНИК ЗАВДАНЬ
# ============================================

class ETLScheduler:
    """
    Планувальник для автоматичного запуску ETL
    """

    def __init__(self, config: ETLConfig):
        self.config = config
        self.logger = logging.getLogger('ETLScheduler')

    def run_daily(self):
        """Запуск щоденного ETL"""
        import schedule
        import time

        def job():
            self.logger.info("Запуск запланованого ETL")
            orchestrator = ETLOrchestrator(self.config)
            orchestrator.run_daily_etl()
```

```
# Планування на кожен день о 18:00
schedule.every().day.at("18:00").do(job)

self.logger.info("Планувальник запущено. Очікування...")

while True:
    schedule.run_pending()
    time.sleep(60)  # Перевірка кожну хвилину
```

```python
# data_quality.py
"""
Модуль для оцінки якості фінансових даних
Версія: 1.0.0
"""

import pandas as pd
import numpy as np
from datetime import datetime
import logging

# Налаштування логування для модуля
logger = logging.getLogger(__name__)

class DataQualityAssessor:
    """
    Клас для оцінки якості даних за 6 критеріями:
    - Повнота (completeness)
    - Точність (accuracy)
    - Узгодженість (consistency)
    - Актуальність (timeliness)
    - Валідність (validity)
    - Унікальність (uniqueness)
    """

    def __init__(self, data_dict):
        """
        Параметри:
        data_dict: словник виду {ticker: DataFrame} де DataFrame має мультиіндексну структуру
                   або звичайний DataFrame з колонками Open, High, Low, Close, Volume
        """
        self.raw_data = data_dict
        self.processed_data = {}
        self.scores = {}

        # Попередня обробка даних
        self._preprocess_data()
        logger.info(f"DataQualityAssessor ініціалізовано з {len(data_dict)} тікерами")

    def _preprocess_data(self):
        """
        Обробка мультиіндексної структури yfinance
        """
        for ticker, df in self.raw_data.items():
            # Створюємо оброблену копію
            processed = pd.DataFrame(index=df.index)

            # Отримуємо Series для кожної колонки
            for col in ['Open', 'High', 'Low', 'Close', 'Volume']:
                if col in df.columns:
                    if isinstance(df[col], pd.DataFrame):
                        # Мультиіндексний випадок
                        processed[col] = df[col].iloc[:, 0]
                    else:
                        # Звичайний DataFrame
                        processed[col] = df[col]
                else:
                    # Якщо колонки немає, створюємо з NaN
                    processed[col] = np.nan

            self.processed_data[ticker] = processed

    def assess_completeness(self):
        """
        Оцінка повноти даних
        """
        results = {}

        for ticker, df in self.processed_data.items():
            # Порівняння торгових і календарних днів
            all_days = pd.date_range(start=df.index.min(), end=df.index.max(), freq='D')
            trading_days = len(df)
            calendar_days = len(all_days)

            # Пропущені дні
            missing_days = calendar_days - trading_days
            missing_pct = (missing_days / calendar_days) * 100

            # Дні з нульовим об'ємом
            if 'Volume' in df.columns:
                volume_series = df['Volume']
                zero_volume = (volume_series == 0).sum()
                zero_volume_pct = (zero_volume / trading_days) * 100
            else:
                zero_volume = 0
                zero_volume_pct = 0

            # Інтегральна оцінка повноти (0-1)
            completeness = 1.0 - (missing_pct / 100) - (zero_volume_pct / 100 * 0.5)
            completeness = max(0, min(1, completeness))
```

```
            results[ticker] = {
                'score': completeness,
                'details': {
                    'trading_days': trading_days,
                    'calendar_days': calendar_days,
                    'missing_days': missing_days,
                    'missing_pct': missing_pct,
                    'zero_volume': zero_volume,
                    'zero_volume_pct': zero_volume_pct
                }
            }

        self.scores['completeness'] = results
        return results

    def assess_accuracy(self):
        """
        Оцінка точності даних
        """
        results = {}

        for ticker, df in self.processed_data.items():
            issues = []
            penalty = 0

            # 1. Логічні перевірки цінових відношень
            if (df['High'] < df['Low']).any():
                issues.append("High < Low")
                penalty += 0.1

            if (df['High'] < df['Close']).any():
                issues.append("High < Close")
                penalty += 0.1

            if (df['Low'] > df['Close']).any():
                issues.append("Low > Close")
                penalty += 0.1

            # 2. Перевірка на негативні ціни
            if (df['Close'] <= 0).any():
                issues.append("Негативні ціни")
                penalty += 0.2

            # 3. Аномальні зміни ціни
            returns = df['Close'].pct_change() * 100
            extreme_returns = (abs(returns) > 20).sum()
            if extreme_returns > 0:
                extreme_pct = (extreme_returns / len(df)) * 100
                issues.append(f"{extreme_returns} днів зі зміною >20% ({extreme_pct:.1f}%)")
                penalty += extreme_pct / 100

            # 4. Z-оцінка для виявлення викидів
            z_scores = np.abs((df['Close'] - df['Close'].mean()) / df['Close'].std())
            extreme_z = (z_scores > 5).sum()
            if extreme_z > 0:
                issues.append(f"{extreme_z} викидів за Z-оцінкою >5σ")
                penalty += extreme_z / len(df) * 2

            # Фінальна оцінка
            accuracy = max(0, 1 - penalty)

            results[ticker] = {
                'score': accuracy,
                'details': {
                    'issues': issues,
                    'penalty': penalty,
                    'extreme_returns': extreme_returns,
                    'extreme_z': extreme_z
                }
            }

        self.scores['accuracy'] = results
        return results

    def assess_consistency(self):
        """
        Оцінка узгодженості даних
        """
        results = {}

        for ticker, df in self.processed_data.items():
            # Аналіз інтервалів між торговими днями
            intervals = df.index.to_series().diff().dt.days
            intervals = intervals.dropna()

            if len(intervals) > 0:
                normal_intervals = intervals.isin([1, 3]).sum()
                consistency_pct = (normal_intervals / len(intervals)) * 100

                # Великі пропуски (>5 днів)
                large_gaps = intervals[intervals > 5]
```

```
                # Оцінка
                consistency = consistency_pct / 100
                if len(large_gaps) > 0:
                    consistency *= (1 - len(large_gaps) / len(intervals))
            else:
                consistency = 1.0
                large_gaps = []
                normal_intervals = 0
                consistency_pct = 100

            results[ticker] = {
                'score': consistency,
                'details': {
                    'min_interval': intervals.min() if len(intervals) > 0 else 0,
                    'max_interval': intervals.max() if len(intervals) > 0 else 0,
                    'avg_interval': intervals.mean() if len(intervals) > 0 else 0,
                    'normal_intervals_pct': consistency_pct,
                    'large_gaps': len(large_gaps),
                    'large_gaps_dates': [d.strftime('%Y-%m-%d') for d in large_gaps.index[:3]] if
len(large_gaps) > 0 else []
                }
            }

        self.scores['consistency'] = results
        return results

    def assess_timeliness(self):
        """
        Оцінка актуальності даних
        """
        results = {}
        current_date = datetime.now()

        for ticker, df in self.processed_data.items():
            last_date = df.index.max()
            days_delay = (current_date - last_date).days

            # Оцінка актуальності
            if days_delay <= 1:
                timeliness = 1.0
                status = "Актуально"
            elif days_delay <= 5:
                timeliness = 0.8
                status = "Невелика затримка"
            elif days_delay <= 20:
                timeliness = 0.5
                status = "Застарілі дані"
            elif days_delay <= 60:
                timeliness = 0.2
                status = "Критично застарілі"
            else:
                timeliness = 0.0
                status = "Архівні дані"

            results[ticker] = {
                'score': timeliness,
                'details': {
                    'last_date': last_date.strftime('%Y-%m-%d'),
                    'days_delay': days_delay,
                    'status': status
                }
            }

        self.scores['timeliness'] = results
        return results

    def assess_validity(self):
        """
        Оцінка валідності даних
        """
        results = {}

        for ticker, df in self.processed_data.items():
            violations = []
            penalty = 0

            # 1. Ціни мають бути додатними
            if (df['Close'] <= 0).any():
                violations.append("Негативні або нульові ціни")
                penalty += 0.2

            # 2. Об'єм має бути невід'ємним
            if 'Volume' in df.columns and (df['Volume'] < 0).any():
                violations.append("Від'ємний об'єм торгів")
                penalty += 0.1

            # 3. Перевірка волатильності
            returns = df['Close'].pct_change().dropna()
            if len(returns) > 0:
                volatility = returns.std() * np.sqrt(252)
```

```
            if volatility > 1.0:
                violations.append(f"Екстремальна волатильність: {volatility:.1%}")
                penalty += min(0.3, volatility / 5)
        else:
            volatility = 0

        # 4. Перевірка на стабільність
        no_change = (df['Close'] == df['Open']).sum()
        no_change_pct = no_change / len(df) if len(df) > 0 else 0
        if no_change_pct > 0.5:
            violations.append(f"Аномально багато днів без змін: {no_change_pct:.1%}")
            penalty += 0.2

        # Фінальна оцінка
        validity = max(0, 1 - penalty)

        results[ticker] = {
            'score': validity,
            'details': {
                'violations': violations,
                'penalty': penalty,
                'volatility': volatility if 'volatility' in locals() else 0,
                'no_change_pct': no_change_pct
            }
        }

    self.scores['validity'] = results
    return results

def assess_uniqueness(self):
    """
    Оцінка унікальності даних
    """
    results = {}

    for ticker, df in self.processed_data.items():
        issues = []
        penalty = 0

        # 1. Дублікати дат
        duplicate_dates = df.index.duplicated().sum()
        if duplicate_dates > 0:
            issues.append(f"{duplicate_dates} дублікатів дат")
            penalty += duplicate_dates / len(df) if len(df) > 0 else 0

        # 2. Повні дублікати рядків
        duplicate_rows = df.duplicated().sum()
        if duplicate_rows > 0:
            issues.append(f"{duplicate_rows} повних дублікатів рядків")
            penalty += (duplicate_rows / len(df)) * 2 if len(df) > 0 else 0

        # 3. Дублікати цін підряд
        price_duplicates = (df['Close'] == df['Close'].shift(1)).sum()
        price_dup_pct = price_duplicates / len(df) if len(df) > 0 else 0
        if price_dup_pct > 0.1:
            issues.append(f"Аномально багато повторів цін: {price_dup_pct:.1%}")
            penalty += price_dup_pct

        # Фінальна оцінка
        uniqueness = max(0, 1 - min(penalty, 1))

        results[ticker] = {
            'score': uniqueness,
            'details': {
                'issues': issues,
                'penalty': penalty,
                'duplicate_dates': duplicate_dates,
                'duplicate_rows': duplicate_rows,
                'price_duplicates_pct': price_dup_pct
            }
        }

    self.scores['uniqueness'] = results
    return results

def comprehensive_report(self):
    """
    Повний звіт про якість даних
    """
    # Виконуємо всі оцінки
    self.assess_completeness()
    self.assess_accuracy()
    self.assess_consistency()
    self.assess_timeliness()
    self.assess_validity()
    self.assess_uniqueness()

    # Створюємо зведений DataFrame
    report_data = []
    for ticker in self.processed_data.keys():
        row = {'Ticker': ticker}
```

```
            for criterion in ['completeness', 'accuracy', 'consistency',
                              'timeliness', 'validity', 'uniqueness']:
                criterion_name = criterion.capitalize()
                if criterion in self.scores and ticker in self.scores[criterion]:
                    row[criterion_name] = self.scores[criterion][ticker]['score']
                else:
                    row[criterion_name] = 0.0

            # Загальна оцінка
            scores = [row[c.capitalize()] for c in ['completeness', 'accuracy', 'consistency',
                                                    'timeliness', 'validity', 'uniqueness']]
            row['Overall'] = np.mean(scores)

            # Рекомендація
            if row['Overall'] < 0.5:
                row['Recommendation'] = '✗ НЕ ПРИДАТНІ'
            elif row['Overall'] < 0.7:
                row['Recommendation'] = '⚠ Обмежене використання'
            elif row['Overall'] < 0.9:
                row['Recommendation'] = '✓ Придатні'
            else:
                row['Recommendation'] = '★ ВІДМІННА якість'

            report_data.append(row)

        report_df = pd.DataFrame(report_data)
        if 'Ticker' in report_df.columns:
            report_df = report_df.set_index('Ticker')

        return report_df

    def detailed_report(self, ticker):
        """
        Детальний звіт для конкретного тікера
        """
        if ticker not in self.processed_data:
            return f"Тікер {ticker} не знайдено"

        print(f"\n{'='*60}")
        print(f"ДЕТАЛЬНИЙ ЗВІТ ПРО ЯКІСТЬ ДАНИХ: {ticker}")
        print(f"{'='*60}")

        for criterion in ['completeness', 'accuracy', 'consistency',
                          'timeliness', 'validity', 'uniqueness']:
            if criterion in self.scores and ticker in self.scores[criterion]:
                data = self.scores[criterion][ticker]
                print(f"\n► {criterion.upper()}: {data['score']:.1%}")

                if 'details' in data:
                    for key, value in data['details'].items():
                        if isinstance(value, list):
                            if value:
                                print(f"  • {key}: {', '.join(str(v) for v in value[:3])}")
                                if len(value) > 3:
                                    print(f"    ... та ще {len(value) - 3}")
                        elif isinstance(value, float):
                            print(f"  • {key}: {value:.2f}")
                        else:
                            print(f"  • {key}: {value}")

# Функція для зручного використання
def assess_data_quality(data_dict):
    """
    Зручна функція для швидкої оцінки якості даних
    """
    assessor = DataQualityAssessor(data_dict)
    return assessor.comprehensive_report()

# Версія модуля
__version__ = "1.0.0"
```

Практичні завдання

1. Вивчіть наведений код реалізації ETL-pipeline. Які типи подій фіксуються логером і які типи повідомлень можуть надходити до файлу etl_pipeline.log? Наведіть приклади з коду реалізації ETL-pipeline по кожному такому типу, а також приклади наявних записів у log-файлі.
2. Виконайте заміну тікерів в коді прикладу на тікери свого варіанту та виконайте запуск. Файл data_quality.py має бути розташований там, де і код ETL-pipeline.
3. Для чого призначено клас DataWarehouseAnalytics? Які функції він реалізує?
4. Виконайте дослідження ETL-процесу і визначіть час завантаження даних по кожному тікеру. Яка успішність завантаження по кожному із тікерів та відсоток втрати даних при завантажені?
5. Скільки записів у кожній таблиці? Напишіть SQL-запит для отримання всіх даних по певному тікеру. Перевірте цілістність даних та відсутність дублікатів.
6. Змініть період завантаження на 2 роки (730 днів), запустіть ETL і порівняйте час виконання.
7. Підготуйте звіт з виконання практичних завдань.