ChatGPT

Промт для генерации кода PipelineBase

Напиши Python-код абстрактного базового класса `PipelineBase` для проекта **BioETL**, который реализует единый табличный ETL-интерфейс. Класс должен следовать шаблону **Extract → Transform → Validate → Write**. Включи следующие требования:

- **Методы класса** (с нужными аннотациями типов):
- `extract(*args, **kwargs) -> pd.DataFrame` – **абстрактный метод**. Отвечает за извлечение данных из внешнего источника. Должен быть реализован в подклассах.
- `transform(df: pd.DataFrame) -> pd.DataFrame` – **абстрактный метод**. Отвечает за преобразование/очистку данных. Реализуется в подклассах.
- `validate(df: pd.DataFrame) -> pd.DataFrame` – метод валидации данных с поддержкой схемы **Pandera**/**Pydantic**. Если у объекта `self.schema` задана схема (например, `pandera.DataFrameSchema`), метод должен проверить `df` на соответствие схеме. После успешной валидации нужно отсортировать столбцы `df` согласно определению схемы для консистентности. Если `self.schema` не задан, метод просто возвращает исходный `df`.
- `write(df: pd.DataFrame, output_path: Path) -> WriteResult` – метод записи результатов на диск. Должен отсортировать DataFrame (например, по первичному ключу, если он задан в `config`) перед сохранением. Записывает отсортированные данные (например, в CSV) и генерирует файл с метаданными (`meta.yaml`) о выполнении. Метаданные могут включать время запуска, `pipeline_name`, `run_id`, количество записей и т.д. Путь для `meta.yaml` формируется с учётом `output_path`, имени пайплайна и идентификатора запуска (например: `<output_path>/<pipeline_name>/<run_id>/meta.yaml`). Метод возвращает объект `WriteResult` (например, с информацией об итоговом файле и количестве записанных записей).
- `run(output_path: Path, *args, dry_run: bool = False, **kwargs) -> RunResult` – основной метод запуска конвейера ETL. Последовательно выполняет стадии: `extract`, `transform`, `validate`. Если `dry_run=False`, выполняет затем стадию `write` для сохранения результатов. Если `dry_run=True`, этап записи пропускается. Метод возвращает объект `RunResult`, содержащий информацию о результате запуска (например, статус, путь вывода, количество обработанных записей и т.д.). По завершении (в обоих случаях) следует освободить внешние ресурсы (см. `close_resources()`).
- `register_client(name: str, client: Any) -> None` – регистрирует внешний ресурс (например, клиент API или подключение к БД) под именем `name`. Сохраняет объект клиента во внутреннем словаре `_clients` для последующего использования.
- `close_resources() -> None` – закрывает все зарегистрированные внешние ресурсы. Проходит по `_clients` и для каждого клиента вызывает метод закрытия соединения (например, `.close()` или `.disconnect()`, если такие имеются). После закрытия очищает словарь `_clients`.

- `_add_hash_columns(df: pd.DataFrame) -> pd.DataFrame` – приватный вспомогательный метод. Добавляет в DataFrame одну или несколько колонок-хэшей, например вычисляет хэш каждой строки (например, MD5 от конкатенации значений) для отслеживания изменений данных. Возвращает обновленный DataFrame. (Этот метод **не**

обязателен к переопределению, предоставляет общую реализацию для использования при необходимости.)

- **Свойства класса**:

- `config: PipelineConfig` – объект конфигурации пайплайна (например, содержит параметры подключения, имя сущности, первичный ключ и т.д.).
- `run_id: str` – уникальный идентификатор текущего запуска (может генерироваться автоматически, например UUID или метка времени, если не задан явно).
- `pipeline_name: str` – имя (код) пайплайна. Можно задавать на основе `config` (например, поле `name` в конфигурации) или имени класса. Используется при логировании и формировании путей для вывода.
- `_clients: dict[str, Any]` – внутренний словарь зарегистрированных клиентов/ресурсов, индексированных по именам.

- `_logger` – объект логирования (например, на базе `logging.Logger`), для записи хода выполнения этапов пайплайна.

- **Поддержка схемы (Pandera/Pydantic)**: Внутри класса предусмотреть свойство `self.schema` (или использование `self.config`, если схема описана там) для хранения схемы данных. Метод `validate` должен использовать эту схему:

- Если используется **Pandera**, предполагается объект `pandera.DataFrameSchema` для валидации всего DataFrame. После применения `schema.validate(df)` получить обратно проверенный DataFrame. Также обеспечить упорядочение колонок в соответствии с определением `DataFrameSchema` (например, используя порядок ключей `schema.columns`).
- Если используется **Pydantic** для описания схемы (например, модель для отдельных записей), можно валидировать каждую строку через Pydantic-модель, однако по умолчанию в базовом классе можно ограничиться поддержкой Pandera для всей таблицы сразу (Pydantic-схемы могут быть обёрнуты отдельно при необходимости).

- Если схема не задана, `validate` просто возвращает данные без изменений.

- **Метаданные о запуске**: При записи результатов реализовать создание файла мета-информации (например, `meta.yaml`). Путь файла метаданных составляется на основе `output_path`, имени пайплайна и `run_id`. В метаданные включить ключевую информацию о запуске: `pipeline_name`, `run_id`, дата/время запуска, количество записанных строк и т.д. Этот файл поможет отслеживать историю запусков и параметры каждого запуска.

- **Режим** `dry_run`: Учесть параметр `dry_run` в методе `run`. Если `dry_run=True`, выполнить только этапы `extract`, `transform` и `validate`, **не выполняя** этап `write`. При этом можно логировать или возвращать `RunResult` с указанием, что запуск был тестовым. Если `dry_run=False`, выполнить полный цикл включая запись на диск.

- **Обогащение данных (enrichment)**: В базовом классе не выполняется никаких дополнительных шагов обогащения данных помимо заданных этапов. Если необходимо доменное обогащение (например, добавление внешних данных или вычисление дополнительных показателей), это должно быть реализовано опционально в конкретных

подклассах (например, путем переопределения метода `transform` или добавления собственных методов). Иначе говоря, в `PipelineBase` этап обогащения по умолчанию **отключён** и никак не влияет на работу пайплайна.

- **Формат данных**: Все стадии конвейера оперируют с данными в формате `pandas.DataFrame`. То есть, `extract` возвращает DataFrame, `transform` принимает и возвращает DataFrame, `validate` работает с DataFrame, и `write` получает DataFrame на вход. Это унифицированный интерфейс для табличных данных в рамках ETL. Убедись, что во всех методах используются аннотации типов `pd.DataFrame` для ясности.

Кроме кода класса `PipelineBase`, предоставь:

- Развернутые **docstring**-комментарии для самого класса и всех его методов, объясняющие их назначение. В docstring класса укажи, что он абстрактный и какие стадии реализует, а в docstring методов опиши их роль (и отметь, какие должны переопределяться в наследниках).
- Минимальный пример наследника, например класс `ChemblTestItemPipeline`, который наследуется от `PipelineBase`. В этом классе:
- **Не используется обогащение** (т.е. нет дополнительных стадий кроме обязательных).
- Реализованы собственные методы `extract()` и `transform()` (для примера можно вернуть небольшой DataFrame с фиктивными данными в `extract` и выполнить простую трансформацию в `transform`).
- Методы `validate` и `write` в примере можно использовать напрямую из `PipelineBase` (если нужна схема для примера, можно задать `self.schema = None` или имитировать простую схему Pandera).
- Добавь docstring для класса наследника и комментарии в коде, чтобы показать, что этот пример предназначен для демонстрации использования базового класса.

Важно: Генерируемый код должен соответствовать стандарту PEP8, содержать необходимые `import`-ы (например, `pandas`, `pathlib.Path`, `uuid`, `logging`, `yaml`, `pandera` и т.д.), а также использовать аннотации типов и строки документирования (docstrings) на русском языке.
