PyPiper Streaming Data Framework

Pypiper is a streaming data framework for Kafka and Flink. It is designed to be a lightweight, easy-to-use, and flexible framework for building streaming data pipelines.

In this repository, you will find the code for the Pypiper framework, as well as examples and documentation.

Schema Models

Below is a detailed walkthrough of the code, its components, and how all the pieces fit together.

Think of it as a step-by-step explanation from the bottom up:

1. Date/Time Helpers & Converters

`python_datetime_to_millis(dt: datetime) -> int`

- Converts a Python datetime object to the number of milliseconds since the Unix epoch (1970-01-01 00:00:00 UTC).
- It calculates the difference (delta) between the given datetime and the epoch, then multiplies total seconds by 1000 to get milliseconds.

`python_time_to_millis(t: time) -> int`

- Converts a Python time object (hours, minutes, seconds, microseconds) into the total number of milliseconds since midnight.
- It computes the total number of seconds (and fractional seconds) and multiplies by 1000.

`python_date_to_days(d: date) -> int`

- Converts a Python date object to the number of days since Unix epoch (1970-01-01).
- Simply subtracts the epoch date from the provided date.

`millis_to_python_datetime(ms: int) -> datetime`

- Converts an integer representing milliseconds since the Unix epoch back to a Python datetime object.

`millis_to_python_time(ms: int) -> time`

- Converts an integer representing milliseconds since midnight into a Python time object.

`days_to_python_date(days: int) -> date`

- Converts an integer representing days since the Unix epoch back into a Python date object.

2. Decimal Helpers

`quantize_decimal(value: Decimal, precision: int, scale: int) -> Decimal`

- Ensures a Decimal value conforms to a specific precision (total number of digits) and scale (digits after the decimal point).
- Uses the quantize method with rounding mode ROUND_HALF_EVEN to standardize the number of digits.
- Raises an error if the value's digits exceed the specified precision or if conversion fails (e.g. DecimalException).

3. Enum Converter

`enum_converter(x, symbols, enum_type)`

- Converts x into a member of a given enumeration (enum_type).
- Supports several possible input types:
    1. A string that matches an enum member name.
    2. An integer index referencing symbols.
    3. Already an enum instance.
- Raises ValueError if x does not map correctly to the enum.

4. Other Converter Functions

These helper functions are used to unify Python↔Avro transformations:

- `decimal_converter`: Converts any input into a Decimal of a specific precision and scale by calling `quantize_decimal`.
- `datetime_converter_py_to_avro` / `datetime_converter_avro_to_py`: Handle conversions between Python datetime and Avro-friendly millisecond timestamps.
- `time_converter_py_to_avro` / `time_converter_avro_to_py`: Handle conversions between Python time and Avro-friendly millisecond times.
- `date_converter_py_to_avro` / `date_converter_avro_to_py`: Handle conversions between Python date and Avro-friendly "days since epoch".
- `record_converter`: Checks if the schema has a deserialize method. If yes, it calls that for nested record deserialization.
- `CONVERTERS`: A dictionary that maps known "logicalType" strings to a converter function. For example, "decimal" -> `decimal_converter`, "time-millis" -> `time_converter_py_to_avro`, etc.

5. SchemaField Class

```
@attrs.define
class SchemaField:
    ...
```

This class acts like a specialized `attrs.field` but adds extra Avro/metadata features.
Attributes:

- `alias`: An optional alternate field name for Avro serialization.

- `logical_type`: Avro logical type (e.g., "decimal", "time-millis", etc.).
- `precision`, `scale`: For decimal fields.
- `default`: Default value if not provided.
- `enum_symbols`, `enum_type`: For enumerations.
- `size`: For Avro fixed types.
- `items`, `values`: For arrays and maps in Avro.

The key method is `__call__`, which returns an `attrs.field` with:

- A converter function (if logical_type is recognized in CONVERTERS).
- Additional metadata for Avro generation (e.g. size, items, etc.).

Essentially, `SchemaField()` is a helper that generates a proper `attrs.field(...)` with Avro-friendly metadata and converter logic baked in.

6. Generating an Avro Schema

```
def generate_avro_schema(cls: Type[Any]) -> Optional[Dict[str, Any]]:
    ...
```

- Builds an Avro schema (as a dictionary) by inspecting each field of the attrs-defined class.
- Checks cls.SchemaConfig to confirm the schema type is "avro".
- Constructs:
    1. name
    2. type: typically "record"
    3. namespace
    4. doc
    5. fields: A list of field definitions. Each field in the list has:
        - "name": The alias if provided, else the field's name.
        - "type": The Avro type (from infer_avro_type).
        - "flink.type": The corresponding Flink SQL type (from infer_flink_type).

If `cls.SchemaConfig._type` is not "avro", it raises an error.

7. Inferring Avro and Flink Types

Two closely related functions:

`infer_avro_type(py_type: Type[Any], metadata: Dict[str, Any]) -> Any`

- Examines a Python type plus metadata (like logicalType, size, items, etc.) to produce the correct Avro type descriptor.
- Handles logical types ("decimal", "time-millis", "enum", etc.) by returning a structured dictionary, e.g.:

```
{
    "type": "bytes",
    "logicalType": "decimal",
    "precision": 10,
```

```
    "scale": 2
}
```

- Handles arrays, maps, and nested records (i.e., if py_type is an attrs class, it calls generate_avro_schema recursively).
- Handles Union type hints by returning a list of possible Avro types (Avro union).

`infer_flink_type(py_type: Type[Any], metadata: Dict[str, Any]) -> str`

- Similar to infer_avro_type but returns a string representing the Flink SQL type (e.g., `DECIMAL(10, 2)`, `ARRAY<STRING>`, `ROW<...>`, etc.).
- Also handles logical types ("decimal", "timestamp-millis", "enum", etc.) by returning the appropriate Flink-compatible type.

8. SchemaConfig Class

```
class SchemaConfig:
    _type: Literal["avro"] = "avro"
    name: str = "DefaultSchema"
    namespace: str = ""
    doc: str = ""
    type: str = "record"
```

- A simple class that holds schema-related information:
  - type identifies the kind of schema (here, forced to "avro").
  - name, namespace, doc, and type shape the Avro schema's basic structure.

9. AvroSchema Base Class

```
class AvroSchema(Generic[T]):
    def to_dict(...):
        ...
    @classmethod
    def from_dict(...):
        ...
```

- A generic base with to_dict and from_dict placeholders.
- The actual logic is implemented by the decorator schema(...) (or AvroModel(...)).

10. AvroModel Decorator

```
def AvroModel(cls: Type[T]) -> Type[AvroSchema]:
    return schema(cls)
```

- Shortcut decorator. Calls schema(cls) to convert an existing class into an Avro-enabled attrs class.

```
def schema(cls: Type[T]) -> Type[AvroSchema]:
    cls = attrs.define(cls)
    if not hasattr(cls, "SchemaConfig"):
        cls.SchemaConfig = SchemaConfig()  # default
    cls.SchemaConfig._type = "avro"

    cls.schema_dict = property(lambda self: generate_avro_schema(cls))
    cls.schema_str = property(lambda self: json.dumps(self.schema_dict))
    cls.schema = property(lambda self: Schema(self.schema_str, "AVRO"))

    ...
    return cls
```

Main Steps Inside `schema(cls)`:

1. Make the class an attrs-defined class: `cls = attrs.define(cls)`.
2. Set or confirm `SchemaConfig`: If the class doesn't have a `SchemaConfig`, one is created. `_type` is forced to `"avro"`.
3. Expose schema properties:
    - `schema_dict`: Generates Avro schema dictionary (via `generate_avro_schema`).
    - `schema_str`: Serializes the schema dictionary to JSON.
    - `schema`: Constructs a `confluent_kafka.schema_registry.Schema` object from `schema_str`.
4. Implement `to_dict`:
    - Loops over each attrs field in the class.
    - Uses the field's `logicalType` to format values in a way Avro expects (e.g., converting datetime to milliseconds).
    - Returns a dictionary appropriate for Avro serialization.
5. Implement `from_dict`:
    - Class method that reconstructs an instance from an Avro-like dictionary.
    - For each field, uses the `logicalType` to convert Avro data back into Python objects (e.g. milliseconds -> datetime, days -> date, etc.).
6. Implement `to_bytes`:
    - Serializes the instance to bytes using Confluent's `AvroSerializer`.
    - Calls `generate_avro_schema` to get the schema.
    - Calls `self_.to_dict()` to get the data.
    - Special handling for Decimal fields: uses `decimal_to_bytes(value, precision, scale)` to produce a byte array in big-endian format.
    - Passes it all to `AvroSerializer(...)` with the given subject.
7. Implement `from_bytes`:
    - Class method to deserialize Avro-encoded bytes back into an instance of the class using Confluent's `AvroDeserializer`.
    - Builds a `from_dict` callback that is handed to `AvroDeserializer`.

When the decorator finishes, the original class has all these attributes and methods added, effectively becoming a complete "Avro-enabled" data model that supports round-trip serialization/deserialization (`Avro <-> Python` objects).

11. `decimal_to_bytes(value: Decimal, precision: int, scale: int) -> bytes`

- Helper function specifically for Avro decimal fields.
- First ensures the value is quantized to the desired precision and scale.
- Then converts the quantized value's integer portion to a signed big-endian byte array.
- Raises a `ValueError` if the value doesn't fit within the specified precision/scale.

Putting It All Together

1. Decorating a Class

When you write:

```
@AvroModel
class MyRecord:
    my_field = SchemaField(
        logical_type="decimal",
        precision=10,
        scale=2,
        default=Decimal("0.0")
    )()
    ...
```

- MyRecord is turned into an attrs class with special Avro-related methods.
- SchemaField(...)() builds an attrs.field that knows how to handle decimal at runtime.
- MyRecord.SchemaConfig ensures `_type="avro"`.

2. Schema Generation

Calling `MyRecord.schema_dict` will produce an Avro schema dictionary. For example:

```
{
  "name": "DefaultSchema",
  "type": "record",
  "namespace": null,
  "doc": null,
  "fields": [
    {
      "name": "my_field",
      "type": {
        "type": "bytes",
        "logicalType": "decimal",
        "precision": 10,
        "scale": 2
      },
```

```
        "flink.type": "DECIMAL(10, 2)"
      }
      ...
    ]
}
```

3. Serialization

- Calling `my_record_instance.to_bytes(registry_client, "my_subject")`:
    1. Generates or loads the Avro schema into Confluent's schema registry if not already present.
    2. Converts the Python object's fields to Avro-compatible data (dict).
    3. Serializes to Avro-format bytes.

4. Deserialization

- Calling `MyRecord.from_bytes(serialized_bytes, registry_client, "my_subject")`:
    1. Pulls the schema from Confluent's schema registry.
    2. Uses the Avro deserializer to decode bytes into a dict.
    3. Converts Avro data (millis, days, decimal bytes) back to Python data structures.
    4. Returns a MyRecord instance.

Overall, the entire module creates a structured, attrs-based Python data model class that integrates with Confluent Kafka's Avro serialization mechanism. It includes Flink SQL type annotations in the generated Avro schema, easing the process of keeping Python, Avro, Kafka, and Flink in sync when exchanging data between services.