Patito

Jakob Gerhard Martinussen

Dec 15, 2022

CONTENTS

CHAPTER

ONE

TUTORIALS

1.1 Using Patito for DataFrame Validation

Have you ever found yourself relying on some column of an external data source being non-nullable only to find out *much* later that the assumption proved to be false? What about discovering that a production machine learning model has had a huge performance regression because a new category was introduced to a categorical column? You might not have encountered any of these *exact* scenarios, but perhaps similar ones; they illustrate the necessity of validating your data.

This is a problem encountered in Data & Insight at Oda all the time. A machine learning model might ingest data from a production system that changes frequently, and the author of the model wants to be notified if certain assumptions no longer hold. Or perhaps a data analyst might rely on a pre-processing step that removes all discontinued products from a data set, and this should be validated and communicated clearly in their Jupyter notebook.

At Oda we recently open-sourced patito, a dataframe validation library built on top of polars, which tries to solve this problem. The polars dataframe library has lately been making the rounds among data scientists at Oda, and for good reasons. It can be considered as a total replacement of the well-known pandas library, initially tempting you with its advertised top-notch performance, but then sealing the deal with its intuitive and expressive API. The exact virtues of polars is a topic for another article, but suffice it to say that it is *highly* recommended and it has some great introductory documentation.

The core idea of Patito is that you should define a so-called *"model"* for each of your data sources. A *model* is a declarative python class which describes the general properties of a tabular data set: the names of all the columns, their types, value bounds, and so on… These models can then be used to validate the data sources when they are ingested into your project's data pipeline. In turn, your models become a trustworthy, centralized catalog of all the core facts about your data, facts you can safely rely upon during development.

Enough chit chat, let's get into some technical details! Let's say that your project keeps track of products, and that these products have four core properties:

1. A unique, numeric identifier
2. A name
3. An ideal temperature zone of either `"dry"`, `"cold"`, or `"frozen"`
4. A product demand given as a percentage of the total sales forecast for the next week

In tabular form the data might look something like this.

Table 1: Table 1: Products

product_id	name	temperature_zone	demand_percentage
1	Apple	dry	0.23%
2	Milk	cold	0.61%
3	Ice cubes	frozen	0.01%
…	…	…	…

We now start to model the restrictions we want to put upon our data. In Patito this is done by defining a class which inherits from `patito.Model`, a class which has one *field annotation* for each column in the data. These models should preferably be defined in a centralized place, conventionally `<YOUR_PROJECT_NAME>/models.py`, where you can easily find and refer to them.

Listing 1: project/models.py

```
from typing import Literal

import patito as pt


class Product(pt.Model):
    product_id: int
    name: str
    temperature_zone: Literal["dry", "cold", "frozen"]
    demand_percentage: float
```

Here we have used `typing.Literal` from the standard library in order to specify that `temperature_zone` is not only a `str`, but *specifically* one of the literal values `"dry"`, `"cold"`, or `"frozen"`. You can now use this class to represent a *single specific instance* of a product:

```
>>> Product(product_id=1, name="Apple", temperature_zone="dry", demand_percentage=0.23)
Product(product_id=1, name='Apple', temperature_zone='dry', demand_percentage=0.23)
```

The class also automatically offers input data validation, for instance if you provide an invalid value for `temperature_zone`.

```
>>> Product(product_id=64, name="Pizza", temperature_zone="oven", demand_percentage=0.12)
ValidationError: 1 validation error for Product
temperature_zone
  unexpected value; permitted: 'dry', 'cold', 'frozen' (type=value_error.const;␣
↪given=oven; permitted=('dry', 'cold', 'frozen'))
```

A discerning reader might notice that this looks suspiciously like pydantic's data models, and that is in fact because it is! Patito's model class is built upon pydantic's `pydantic.BaseClass` and therefore offers all of pydantic's functionality. But the difference is that Patito extends pydantic's validation of *singular object instances* to *collections* of the same objects represented as *dataframes*.

Let's take the data presented in Table 1 and represent it as a polars dataframe.

```
>>> import polars as pl

>>> product_df = pl.DataFrame(
...     {
...         "product_id": [1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
...         "name": ["Apple", "Milk", "Ice cubes"],
...         "temperature_zone": ["dry", "cold", "frozen"],
...         "demand_percentage": [0.23, 0.61, 0.01],
...     }
... )
```

We can now use *Product.validate()* in order to validate the content of our dataframe.

```
>>> from project.models import Product
>>> Product.validate(product_df)
None
```

Well, that wasn't really interesting... The validate method simply returns `None` if no errors are found. It is intended as a guard statement to be put before any logic that requires the data to be valid. That way you can rely on the data being compatible with the given model schema, otherwise the `.validate()` method would have raised an exception. Let's try this with invalid data, setting the temperature zone of one of the products to `"oven"`.

```
>>> invalid_product_df = pl.DataFrame(
...     {
...         "product_id": [64, 64],
...         "name": ["Pizza", "Cereal"],
...         "temperature_zone": ["oven", "dry"],
...         "demand_percentage": [0.07, 0.16],
...     }
... )
>>> Product.validate(invalid_product_df)
ValidationError: 1 validation error for Product
temperature_zone
  Rows with invalid values: {'oven'}. (type=value_error.rowvalue)
```

Now we're talking! Patito allows you to define a single class which validates both singular object instances *and* dataframe collections without code duplication!

Patito tries to rely as much as possible on pydantic's existing modelling concepts, naturally extending them to the dataframe domain where suitable. Model fields annotated with `str` will map to dataframe columns stored as `pl.Utf8`, `int` as `pl.Int8`/`pl.Int16`/.../`pl.Int64`, and so on. Field types wrapped in `Optional` allow null values, while bare types do not.

But certain modelling concepts are not applicable in the context of singular object instances, and are therefore necessarily not part of pydantic's API. Take `product_id` as an example, you would expect this column to be unique across all products and duplicates should therefore be considered invalid. In pydantic you have no way to express this, but Patito expands upon pydantic in various ways in order to represent dataframe-related constraints. One of these extensions is the `unique` parameter accepted by `patito.Field`, which allows you to specify that all the values of a given column should be unique.

Listing 2: project/models.py::Product

```
class Product(pt.Model):
    product_id: int = pt.Field(unique=True)
    name: str
    temperature_zone: Literal["dry", "cold", "frozen"]
    demand_percentage: float
```

The `patito.Field` class accepts the same parameters as `pydantic.Field`, but adds additional dataframe-specific constraints documented *here*. In those cases where Patito's built-in constraints do not suffice, you can specify arbitrary

constraints in the form of polars expressions which must evaluate to `True` for each row in order for the dataframe to be considered valid. Let's say we want to make sure that `demand_percentage` sums up to 100% for the entire dataframe, otherwise we might be missing one or more products. We can do this by passing the `constraints` parameter to `patito.Field`.

Listing 3: project/models.py::Product

```
class Product(pt.Model):
    product_id: int = pt.Field(unique=True)
    name: str
    temperature_zone: Literal["dry", "cold", "frozen"]
    demand_percentage: float = pt.Field(constraints=pt.field.sum() == 100.0)
```

Here `patito.field` is an alias for the field column and is automatically replaced with `polars.col("demand_percentage")` before validation. If we now use this improved class to validate `invalid_product_df`, we should detect new errors.

```
>>> Product.validate(invalid_product_df)
ValidationError: 3 validation errors for Product
product_id
  2 rows with duplicated values. (type=value_error.rowvalue)
temperature_zone
  Rows with invalid values: {'oven'}. (type=value_error.rowvalue)
demand_percentage
  2 rows does not match custom constraints. (type=value_error.rowvalue)
```

Patito has now detected that `product_id` contains duplicates and that `demand_percentage` does not sum up to 100%! Several more properties and methods are available on `patito.Model` as outlined *here*; you can for instance generate valid mock dataframes for testing purposes with *Model.examples()*. You can also dynamically construct models with methods such as *Model.select()*, *Model.prefix()*, and *Model.join()*.

CHAPTER

TWO

API REFERENCE

2.1 patito.DataFrame

class patito.**DataFrame**(*data=None*, *columns=None*, *orient=None*)

A sub-class of polars.DataFrame with additional functionality related to Model.

Two different methods are available for constructing model-aware data frames. Assume a simple model with two fields:

```
>>> import patito as pt
>>> class Product(pt.Model):
...     name: str
...     price_in_cents: int
...
```

We can construct a data frame containing products and then associate the `Product` model to the data frame using `DataFrame.set_model`:

```
>>> df = pt.DataFrame({"name": ["apple", "banana"], "price": [25, 61]}).set_model(
...     Product
... )
```

Alternatively, we can use the custom `Product.DataFrame` class which automatically associates the `Product` model to the data frame at instantiation.

```
>>> df = Product.DataFrame({"name": ["apple", "banana"], "price": [25, 61]})
```

The `df` data frame now has a set of model-aware methods such as as *Product.validate*.

2.1.1 Methods

patito.DataFrame.cast

DataFrame.**cast**(*strict=False*)

Cast columns to *dtypes* specified by the associated Patito model.

Parameters

strict (bool) – If set to `False`, columns which are technically compliant with the specified field type, will not be casted. For example, a column annotated with `int` is technically compliant with `pl.UInt8`, even if `pl.Int64` is the default dtype associated with `int`-annotated fields. If

`strict` is set to `True`, the resulting dtypes will be forced to the default dtype associated with each python type.

Returns
A dataframe with columns casted to the correct dtypes.

Return type
DataFrame[*Model*]

Examples

Create a simple model:

```
>>> import patito as pt
>>> import polars as pl
>>> class Product(pt.Model):
...     name: str
...     cent_price: int = pt.Field(dtype=pl.UInt16)
...
```

Now we can use this model to cast some simple data:

```
>>> Product.DataFrame({"name": ["apple"], "cent_price": ["8"]}).cast()
shape: (1, 2)
┌─────────────────────┐
│ name   cent_price   │
│ ---    ---          │
│ str    u16          │
│                     │
│ apple  8            │
└─────────────────────┘
```

patito.DataFrame.derive

`DataFrame.`**`derive`**`()`

Populate columns which have `pt.Field(derived_from=...)` definitions.

If a column field on the data frame model has `patito.Field(derived_from=...)` specified, the given value will be used to define the column. If `derived_from` is set to a string, the column will be derived from the given column name. Alternatively, an arbitrary polars expression can be given, the result of which will be used to populate the column values.

Returns
A new dataframe where all derivable columns are provided.

Return type
DataFrame[*Model*]

Raises
`TypeError` – If the `derived_from` parameter of `patito.Field` is given as something else than a string or polars expression.

Examples

```
>>> import patito as pt
>>> import polars as pl
>>> class Foo(pt.Model):
...     bar: int = pt.Field(derived_from="foo")
...     double_bar: int = pt.Field(derived_from=2 * pl.col("bar"))
...
>>> Foo.DataFrame({"foo": [1, 2]}).derive()
shape: (2, 3)
┌─────┬─────┬────────────┐
│ foo ┆ bar ┆ double_bar │
│ --- ┆ --- ┆ ---        │
│ i64 ┆ i64 ┆ i64        │
╞═════╪═════╪════════════╡
│ 1   ┆ 1   ┆ 2          │
├╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 2   ┆ 2   ┆ 4          │
└─────┴─────┴────────────┘
```

patito.DataFrame.drop

DataFrame.**drop**(*columns=None*)

Drop one or more columns from the dataframe.

If `name` is not provided then all columns *not* specified by the associated patito model, for instance set with *DataFrame.set_model*, are dropped.

Parameters

columns (Union[str, Sequence[str], None]) – A single column string name, or list of strings, indicating which columns to drop. If not specified, all columns *not* specified by the associated dataframe model will be dropped.

Returns

New dataframe without the specified columns.

Return type

DataFrame[*Model*]

Examples

```
>>> import patito as pt
>>> class Model(pt.Model):
...     column_1: int
...
>>> Model.DataFrame({"column_1": [1, 2], "column_2": [3, 4]}).drop()
shape: (2, 1)
┌──────────┐
│ column_1 │
│ ---      │
│ i64      │
```

(continues on next page)

(continued from previous page)

```
| 1         |
|-          |
| 2         |
└───────────┘
```

patito.DataFrame.fill_null

`DataFrame.fill_null`(*value=None*, *strategy=None*, *limit=None*, *matches_supertype=True*)

Fill null values using a filling strategy, literal, or `Expr`.

If `"default"` is provided as the strategy, the model fields with default values are used to fill missing values.

Parameters

- **value** (`Optional[Any]`) – Value used to fill null values.
- **strategy** (`Optional[Literal['forward', 'backward', 'min', 'max', 'mean', 'zero', 'one', 'defaults']]`) – Accepts the same arguments as `polars.DataFrame.fill_null` in addition to `"defaults"` which will use the field's default value if provided.
- **limit** (`Optional[int]`) – The number of consecutive null values to forward/backward fill. Only valid if `strategy` is `"forward"` or `"backward"`.
- **matches_supertype** (`bool`) – Fill all matching supertype of the fill `value`.

Returns

A new dataframe with nulls filled in according to the provided `strategy` parameter.

Return type

DataFrame[*Model*]

Example

```
>>> import patito as pt
>>> class Product(pt.Model):
...     name: str
...     price: int = 19
...
>>> df = Product.DataFrame(
...     {"name": ["apple", "banana"], "price": [10, None]}
... )
>>> df.fill_null(strategy="defaults")
shape: (2, 2)
┌────────┬───────┐
│ name   ┆ price │
│ ---    ┆ ---   │
│ str    ┆ i64   │
╞════════╪═══════╡
│ apple  ┆ 10    │
├╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌┤
│ banana ┆ 19    │
└────────┴───────┘
```

patito.DataFrame.get

DataFrame.**get**(*predicate=None*)

Fetch the single row that matches the given polars predicate.

If you expect a data frame to already consist of one single row, you can use `.get()` without any arguments to return that row.

Raises

- **RowDoesNotExist** – If zero rows evaluate to true for the given predicate.
- **MultipleRowsReturned** – If more than one row evaluates to true for the given predicate.
- **RuntimeError** – The superclass of both `RowDoesNotExist` and `MultipleRowsReturned` if you want to catch both exceptions with the same class.

Parameters

predicate (`Optional[Expr]`) – A polars expression defining the criteria of the filter.

Returns

A pydantic-derived base model representing the given row.

Return type

Model

Example

```
>>> import patito as pt
>>> import polars as pl
>>> df = pt.DataFrame({"product_id": [1, 2, 3], "price": [10, 10, 20]})
```

The `.get()` will by default return a dynamically constructed pydantic model if no model has been associated with the given dataframe:

```
>>> df.get(pl.col("product_id") == 1)
UntypedRow(product_id=1, price=10)
```

If a Patito model has been associated with the dataframe, by the use of *DataFrame.set_model()*, then the given model will be used to represent the return type:

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     price: float
...
>>> df.set_model(Product).get(pl.col("product_id") == 1)
Product(product_id=1, price=10.0)
```

You can invoke `.get()` without any arguments on dataframes containing exactly one row:

```
>>> df.filter(pl.col("product_id") == 1).get()
UntypedRow(product_id=1, price=10)
```

If the given predicate matches multiple rows a `MultipleRowsReturned` will be raised:

```
>>> try:
...     df.get(pl.col("price") == 10)
... except pt.exceptions.MultipleRowsReturned as e:
...     print(e)
...
DataFrame.get() yielded 2 rows.
```

If the given predicate matches zero rows a `RowDoesNotExist` will be raised:

```
>>> try:
...     df.get(pl.col("price") == 0)
... except pt.exceptions.RowDoesNotExist as e:
...     print(e)
...
DataFrame.get() yielded 0 rows.
```

patito.DataFrame.read_csv

classmethod DataFrame.**read_csv**(**args*, ***kwargs*)

Read CSV and apply correct column name and types from model.

If any fields have `derived_from` specified, the given expression will be used to populate the given column(s).

Parameters

- ***args** – All positional arguments are forwarded to `polars.read_csv`.
- ****kwargs** – All keyword arguments are forwarded to `polars.read_csv`.

Returns

A dataframe representing the given CSV file data.

Return type

DataFrame[*Model*]

Examples

The `DataFrame.read_csv` method can be used to automatically set the correct column names when reading CSV files without headers.

```
>>> import io
>>> import patito as pt
>>> class CSVModel(pt.Model):
...     a: float
...     b: str
...
>>> csv_file = io.StringIO("1,2")
>>> CSVModel.DataFrame.read_csv(csv_file, has_header=False)
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ f64 ┆ str │
```

(continues on next page)

(continued from previous page)

```
│ 1.0  2   │
└──────────┘
```

The `derived_from` paramater of `pt.Field` allows you to specify the mapping between the CSV file's column names, and the final column names you intend to construct.

```
>>> import io
>>> import patito as pt
>>> class CSVModel(pt.Model):
...     a: float
...     b: str = pt.Field(derived_from="source_of_b")
...
>>> csv_file = io.StringIO("a,source_of_b\n1,1")
>>> CSVModel.DataFrame.read_csv(csv_file).drop()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ f64 ┆ str │
╞═════╪═════╡
│ 1.0 ┆ 1   │
└─────┴─────┘
```

patito.DataFrame.set_model

DataFrame.**set_model**(*model*)

Associate a given patito `Model` with the dataframe.

The model schema is used by methods that depend on a model being associated with the given dataframe such as *DataFrame.validate()* and *DataFrame.get()*.

`DataFrame(...).set_model(Model)` is equivalent with `Model.DataFrame(...)`.

Parameters

model (Model) – Sub-class of `patito.Model` declaring the schema of the dataframe.

Returns

Returns the same dataframe, but with an attached model that is required for certain model-specific dataframe methods to work.

Return type

DataFrame[*Model*]

Examples

```
>>> from typing_extensions import Literal
>>> import patito as pt
>>> import polars as pl
>>> class SchoolClass(pt.Model):
...     year: int = pt.Field(dtype=pl.UInt16)
...     letter: Literal["A", "B"] = pt.Field(dtype=pl.Categorical)
...
```

(continues on next page)

(continued from previous page)

```
>>> classes = pt.DataFrame(
...     {"year": [1, 1, 2, 2], "letter": list("ABAB")}
... ).set_model(SchoolClass)
>>> classes
shape: (4, 2)
┌──────┬────────┐
│ year ┆ letter │
│ ---  ┆ ---    │
│ i64  ┆ str    │
╞══════╪════════╡
│ 1    ┆ A      │
├╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌┤
│ 1    ┆ B      │
├╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌┤
│ 2    ┆ A      │
├╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌┤
│ 2    ┆ B      │
└──────┴────────┘
>>> casted_classes = classes.cast()
>>> casted_classes
shape: (4, 2)
┌──────┬────────┐
│ year ┆ letter │
│ ---  ┆ ---    │
│ u16  ┆ cat    │
╞══════╪════════╡
│ 1    ┆ A      │
├╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌┤
│ 1    ┆ B      │
├╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌┤
│ 2    ┆ A      │
├╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌┤
│ 2    ┆ B      │
└──────┴────────┘
>>> casted_classes.validate()
```

patito.DataFrame.validate

DataFrame.**validate**()

Validate the schema and content of the dataframe.

You must invoke `.set_model()` before invoking `.validate()` in order to specify how the dataframe should be validated.

Returns

The original dataframe, if correctly validated.

Return type

DataFrame[*Model*]

Raises

- **TypeError** – If `DataFrame.set_model()` has not been invoked prior to validation. Note that `patito.Model.DataFrame` automatically invokes `DataFrame.set_model()` for you.

- **patito.exceptions.ValidationError** – If the dataframe does not match the specified schema.

Examples

```
>>> import patito as pt
```

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     temperature_zone: Literal["dry", "cold", "frozen"]
...     is_for_sale: bool
...
```

```
>>> df = pt.DataFrame(
...     {
...         "product_id": [1, 1, 3],
...         "temperature_zone": ["dry", "dry", "oven"],
...     }
... ).set_model(Product)
>>> try:
...     df.validate()
... except pt.ValidationError as exc:
...     print(exc)
...
3 validation errors for Product
is_for_sale
  Missing column (type=type_error.missingcolumns)
product_id
  2 rows with duplicated values. (type=value_error.rowvalue)
temperature_zone
  Rows with invalid values: {'oven'}. (type=value_error.rowvalue)
```

2.2 patito.Model

class patito.**Model**(***data*)

Custom pydantic class for representing table schema and constructing rows.

2.2.1 Class properties

patito.Model.DataFrame

Model.**DataFrame**(*data=None*, *columns=None*, *orient=None*)

A sub-class of polars.DataFrame with additional functionality related to Model.

Two different methods are available for constructing model-aware data frames. Assume a simple model with two fields:

```
>>> import patito as pt
>>> class Product(pt.Model):
...     name: str
...     price_in_cents: int
...
```

We can construct a data frame containing products and then associate the `Product` model to the data frame using `DataFrame.set_model`:

```
>>> df = pt.DataFrame({"name": ["apple", "banana"], "price": [25, 61]}).set_model(
...     Product
... )
```

Alternatively, we can use the custom `Product.DataFrame` class which automatically associates the `Product` model to the data frame at instantiation.

```
>>> df = Product.DataFrame({"name": ["apple", "banana"], "price": [25, 61]})
```

The `df` data frame now has a set of model-aware methods such as as *Product.validate*.

patito.Model.LazyFrame

`Model.`**`LazyFrame`**`()`

LazyFrame class associated to DataFrame.

patito.Model.columns

property `Model.`**`columns: List[str]`**

Return the name of the dataframe columns specified by the fields of the model.

Return type
`List[str]`

Returns
List of column names.

Example

```
>>> import patito as pt
>>> class Product(pt.Model):
...     name: str
...     price: int
...
>>> Product.columns
['name', 'price']
```

patito.Model.defaults

property Model.**defaults: dict[str, Any]**

Return default field values specified on the model.

Return type
dict[str, Any]

Returns
Dictionary containing fields with their respective default values.

Example

```
>>> from typing_extensions import Literal
>>> import patito as pt
>>> class Product(pt.Model):
...     name: str
...     price: int = 0
...     temperature_zone: Literal["dry", "cold", "frozen"] = "dry"
...
>>> Product.defaults
{'price': 0, 'temperature_zone': 'dry'}
```

patito.Model.dtypes

property Model.**dtypes: dict[str, Type[polars.datatypes.DataType]]**

Return the polars dtypes of the dataframe.

Unless Field(dtype=...) is specified, the highest signed column dtype is chosen for integer and float columns.

Return type
dict[str, Type[DataType]]

Returns
A dictionary mapping string column names to polars dtype classes.

Example

```
>>> import patito as pt
>>> class Product(pt.Model):
...     name: str
...     ideal_temperature: int
...     price: float
...
>>> Product.dtypes
{'name': <class 'polars.datatypes.Utf8'>, 'ideal_temperature': <class 'polars.
↪datatypes.Int64'>, 'price': <class 'polars.datatypes.Float64'>}
```

patito.Model.non_nullable_columns

property Model.**non_nullable_columns: set[str]**

Return names of those columns that are non-nullable in the schema.

Return type
set[str]

Returns
Set of column name strings.

Example

```
>>> from typing import Optional
>>> import patito as pt
>>> class MyModel(pt.Model):
...     nullable_field: Optional[int]
...     inferred_nullable_field: int = None
...     non_nullable_field: int
...     another_non_nullable_field: str
...
>>> sorted(MyModel.non_nullable_columns)
['another_non_nullable_field', 'non_nullable_field']
```

patito.Model.nullable_columns

property Model.**nullable_columns: set[str]**

Return names of those columns that are nullable in the schema.

Return type
set[str]

Returns
Set of column name strings.

Example

```
>>> from typing import Optional
>>> import patito as pt
>>> class MyModel(pt.Model):
...     nullable_field: Optional[int]
...     inferred_nullable_field: int = None
...     non_nullable_field: int
...     another_non_nullable_field: str
...
>>> sorted(MyModel.nullable_columns)
['inferred_nullable_field', 'nullable_field']
```

patito.Model.sql_types

property Model.**sql_types: dict[str, str]**

Return compatible DuckDB SQL types for all model fields.

Return type
dict[str, str]

Returns
Dictionary with column name keys and SQL type identifier strings.

Example

```
>>> from typing import Literal
>>> import patito as pt
```

```
>>> class MyModel(pt.Model):
...     int_column: int
...     str_column: str
...     float_column: float
...     literal_column: Literal["a", "b", "c"]
...
>>> MyModel.sql_types
{'int_column': 'INTEGER',
 'str_column': 'VARCHAR',
 'float_column': 'DOUBLE',
 'literal_column': 'enum__4a496993dde04060df4e15a340651b45'}
```

patito.Model.unique_columns

property Model.**unique_columns: set[str]**

Return columns with uniqueness constraint.

Return type
set[str]

Returns
Set of column name strings.

Example

```
>>> from typing import Optional
>>> import patito as pt
```

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     barcode: Optional[str] = pt.Field(unique=True)
...     name: str
...
>>> sorted(Product.unique_columns)
['barcode', 'product_id']
```

patito.Model.valid_dtypes

property Model.**valid_dtypes: dict[str, List[Union[Type[polars.datatypes.DataType], polars.datatypes.DataType, polars.datatypes.List]]]**

Return a list of polars dtypes which Patito considers valid for each field.

The first item of each list is the default dtype chosen by Patito.

Return type
dict[str, List[Union[Type[DataType], DataType, List]]]

Returns
A dictionary mapping each column string name to a list of valid dtypes.

Raises
NotImplementedError – If one or more model fields are annotated with types not compatible with polars.

Example

```
>>> from pprint import pprint
>>> import patito as pt
```

```
>>> class MyModel(pt.Model):
...     bool_column: bool
...     str_column: str
...     int_column: int
...     float_column: float
...
>>> pprint(MyModel.valid_dtypes)
{'bool_column': [<class 'polars.datatypes.Boolean'>],
 'float_column': [<class 'polars.datatypes.Float64'>,
                  <class 'polars.datatypes.Float32'>],
 'int_column': [<class 'polars.datatypes.Int64'>,
                <class 'polars.datatypes.Int32'>,
                <class 'polars.datatypes.Int16'>,
                <class 'polars.datatypes.Int8'>,
                <class 'polars.datatypes.UInt64'>,
                <class 'polars.datatypes.UInt32'>,
                <class 'polars.datatypes.UInt16'>,
                <class 'polars.datatypes.UInt8'>],
 'str_column': [<class 'polars.datatypes.Utf8'>]}
```

patito.Model.valid_sql_types

property Model.**valid_sql_types: dict[str, List[Literal['BIGINT', 'INT8', 'LONG', 'BLOB', 'BYTEA', 'BINARY', 'VARBINARY', 'BOOLEAN', 'BOOL', 'LOGICAL', 'DATE', 'DOUBLE', 'FLOAT8', 'NUMERIC', 'DECIMAL', 'HUGEINT', 'INTEGER', 'INT4', 'INT', 'SIGNED', 'INTERVAL', 'REAL', 'FLOAT4', 'FLOAT', 'SMALLINT', 'INT2', 'SHORT', 'TIME', 'TIMESTAMP', 'DATETIME', 'TIMESTAMP WITH TIMEZONE', 'TIMESTAMPTZ', 'TINYINT', 'INT1', 'UBIGINT', 'UINTEGER', 'USMALLINT', 'UTINYINT', 'UUID', 'VARCHAR', 'CHAR', 'BPCHAR', 'TEXT', 'STRING']]]**

Return a list of DuckDB SQL types which Patito considers valid for each field.

The first item of each list is the default dtype chosen by Patito.

Return type
dict[str, List[Literal['BIGINT', 'INT8', 'LONG', 'BLOB', 'BYTEA', 'BINARY', 'VARBINARY', 'BOOLEAN', 'BOOL', 'LOGICAL', 'DATE', 'DOUBLE', 'FLOAT8', 'NUMERIC', 'DECIMAL', 'HUGEINT', 'INTEGER', 'INT4', 'INT', 'SIGNED', 'INTERVAL', 'REAL', 'FLOAT4', 'FLOAT', 'SMALLINT', 'INT2', 'SHORT', 'TIME', 'TIMESTAMP', 'DATETIME', 'TIMESTAMP WITH TIMEZONE', 'TIMESTAMPTZ', 'TINYINT', 'INT1', 'UBIGINT', 'UINTEGER', 'USMALLINT', 'UTINYINT', 'UUID', 'VARCHAR', 'CHAR', 'BPCHAR', 'TEXT', 'STRING']]]

Returns
A dictionary mapping each column string name to a list of DuckDB SQL types represented as strings.

Raises
NotImplementedError – If one or more model fields are annotated with types not compatible with DuckDB.

Example

```
>>> import patito as pt
>>> from pprint import pprint
```

```
>>> class MyModel(pt.Model):
...     bool_column: bool
...     str_column: str
...     int_column: int
...     float_column: float
...
>>> pprint(MyModel.valid_sql_types)
{'bool_column': ['BOOLEAN', 'BOOL', 'LOGICAL'],
 'float_column': ['DOUBLE',
                                    'FLOAT8',
                                    'NUMERIC',
                                    'DECIMAL',
                                    'REAL',
                                    'FLOAT4',
                                    'FLOAT'],
  'int_column': ['INTEGER',
                                 'INT4',
                                 'INT',
                                 'SIGNED',
                                 'BIGINT',
                                 'INT8',
                                 'LONG',
                                 'HUGEINT',
                                 'SMALLINT',
                                 'INT2',
                                 'SHORT',
                                 'TINYINT',
                                 'INT1',
                                 'UBIGINT',
                                 'UINTEGER',
                                 'USMALLINT',
```

(continues on next page)

(continued from previous page)

```
                                    'UTINYINT'],
 'str_column': ['VARCHAR', 'CHAR', 'BPCHAR', 'TEXT', 'STRING']}
```

2.2.2 Class methods

patito.Model.drop

classmethod Model.**drop**(*name*)

Return a new model where one or more fields are excluded.

Parameters

name (Union[str, Iterable[str]]) – A single string field name, or a list of such field names, which will be dropped.

Return type

Type[*Model*]

Returns

New model class where the given fields have been removed.

Examples

```
>>> class MyModel(Model):
...     a: int
...     b: int
...     c: int
...
```

```
>>> MyModel.columns
['a', 'b', 'c']
```

```
>>> MyModel.drop("c").columns
['a', 'b']
```

```
>>> MyModel.drop(["b", "c"]).columns
['a']
```

patito.Model.example

classmethod Model.**example**(***kwargs*)

Produce model instance with filled dummy data for all unspecified fields.

The type annotation of unspecified field is used to fill in type-correct dummy data, e.g. `-1` for `int`, `"dummy_string"` for `str`, and so on…

The first item of `typing.Literal` annotations are used for dummy values.

Parameters

****kwargs** (Any) – Provide explicit values for any fields which should *not* be filled with dummy data.

Returns
A pydantic model object filled with dummy data for all unspecified model fields.

Return type
Model

Raises
TypeError – If one or more of the provided keyword arguments do not match any fields on the model.

Example

```
>>> from typing import Literal
>>> import patito as pt
```

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     name: str
...     temperature_zone: Literal["dry", "cold", "frozen"]
...
>>> Product.example(product_id=1)
Product(product_id=1, name='dummy_string', temperature_zone='dry')
```

patito.Model.example_value

classmethod Model.**example_value**(*field*)

Return a valid example value for the given model field.

Parameters
field (str) – Field name identifier.

Return type
Union[date, datetime, float, int, str, None]

Returns
A single value which is consistent with the given field definition.

Raises
NotImplementedError – If the given field has no example generator.

Example

```
>>> from typing import Literal
>>> import patito as pt
```

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     name: str
...     temperature_zone: Literal["dry", "cold", "frozen"]
...
>>> Product.example_value("product_id")
-1
```

(continues on next page)

(continued from previous page)

```
>>> Product.example_value("name")
'dummy_string'
>>> Product.example_value("temperature_zone")
'dry'
```

patito.Model.examples

classmethod Model.**examples**(*data=None*, *columns=None*)

Generate polars dataframe with dummy data for all unspecified columns.

This constructor accepts the same data format as polars.DataFrame.

Parameters

- **data** (Union[dict, Iterable, None]) – Data to populate the dummy dataframe with. If given as an iterable of values, then column names must also be provided. If not provided at all, a dataframe with a single row populated with dummy data is provided.
- **columns** (Optional[Iterable[str]]) – Ignored if `data` is provided as a dictionary. If data is provided as an `iterable`, then `columns` will be used as the column names in the resulting dataframe. Defaults to None.

Return type

DataFrame

Returns

A polars dataframe where all unspecified columns have been filled with dummy data which should pass model validation.

Raises

TypeError – If one or more of the model fields are not mappable to polars column dtype equivalents.

Example

```
>>> from typing import Literal
>>> import patito as pt
```

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     name: str
...     temperature_zone: Literal["dry", "cold", "frozen"]
...
```

```
>>> Product.examples()
shape: (1, 3)
┌─────────────┬──────────────────┬────────────┐
│ name        ┆ temperature_zone ┆ product_id │
│ ---         ┆ ---              ┆ ---        │
│ str         ┆ cat              ┆ i64        │
╞═════════════╪══════════════════╪════════════╡
│ dummy_string ┆ dry             ┆ 0          │
└─────────────┴──────────────────┴────────────┘
```

```
>>> Product.examples({"name": ["product A", "product B"]})
shape: (2, 3)
┌───────────┬──────────────────┬────────────┐
│ name      ┆ temperature_zone ┆ product_id │
│ ---       ┆ ---              ┆ ---        │
│ str       ┆ cat              ┆ i64        │
╞═══════════╪══════════════════╪════════════╡
│ product A ┆ dry              ┆ 0          │
├╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌┤
│ product B ┆ dry              ┆ 1          │
└───────────┴──────────────────┴────────────┘
```

patito.Model.from_row

classmethod Model.**from_row**(*row*, *validate=True*)

Represent a single data frame row as a Patito model.

Parameters

- **row** (Union[DataFrame, DataFrame]) – A dataframe, either polars and pandas, consisting of a single row.
- **validate** (bool) – If False, skip pydantic validation of the given row data.

Returns

A patito model representing the given row data.

Return type

Model

Raises

TypeError – If the given type is neither a pandas or polars DataFrame.

Example

```
>>> import patito as pt
>>> import polars as pl
```

```
>>> class Product(pt.Model):
...     product_id: int
...     name: str
...     price: float
...
```

```
>>> df = pl.DataFrame(
...     [["1", "product name", "1.22"]],
...     columns=["product_id", "name", "price"],
... )
>>> Product.from_row(df)
Product(product_id=1, name='product name', price=1.22)
>>> Product.from_row(df, validate=False)
Product(product_id='1', name='product name', price='1.22')
```

patito.Model.join

classmethod Model.**join**(*other*, *how*)

Dynamically create a new model compatible with an SQL Join operation.

For instance, `ModelA.join(ModelB, how="left")` will create a model containing all the fields of `ModelA` and `ModelB`, but where all fields of `ModelB` has been made `Optional`, i.e. nullable. This is consistent with the LEFT JOIN SQL operation making all the columns of the right table nullable.

Parameters

- **other** (Type[*Model*]) – Another patito Model class.
- **how** (Literal['inner', 'left', 'outer', 'asof', 'cross', 'semi', 'anti']) – The type of SQL Join operation.

Return type

Type[*Model*]

Returns

A new model type compatible with the resulting schema produced by the given join operation.

Examples

```
>>> class A(Model):
...     a: int
...
>>> class B(Model):
...     b: int
...
```

```
>>> InnerJoinedModel = A.join(B, how="inner")
>>> InnerJoinedModel.columns
['a', 'b']
>>> InnerJoinedModel.nullable_columns
set()
```

```
>>> LeftJoinedModel = A.join(B, how="left")
>>> LeftJoinedModel.nullable_columns
{'b'}
```

```
>>> OuterJoinedModel = A.join(B, how="outer")
>>> sorted(OuterJoinedModel.nullable_columns)
['a', 'b']
```

```
>>> A.join(B, how="anti") is A
True
```

patito.Model.pandas_examples

classmethod Model.**pandas_examples**(*data*, *columns=None*)

Generate dataframe with dummy data for all unspecified columns.

Offers the same API as the pandas.DataFrame constructor. Non-iterable values, besides strings, are repeated until they become as long as the iterable arguments.

Parameters

- **data** (Union[dict, Iterable]) – Data to populate the dummy dataframe with. If not a dict, column names must also be provided.
- **columns** (Optional[Iterable[str]]) – Ignored if data is a dict. If data is an iterable, it will be used as the column names in the resulting dataframe. Defaults to None.

Return type
DataFrame

Returns
A pandas DataFrame filled with dummy example data.

Raises

- **ImportError** – If pandas has not been installed. You should install patito[pandas] in order to integrate patito with pandas.
- **TypeError** – If column names have not been specified in the input data.

Example

```
>>> from typing import Literal
>>> import patito as pt
```

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     name: str
...     temperature_zone: Literal["dry", "cold", "frozen"]
...
```

```
>>> Product.pandas_examples({"name": ["product A", "product B"]})
   product_id       name temperature_zone
0          -1  product A              dry
1          -1  product B              dry
```

patito.Model.prefix

classmethod Model.**prefix**(*prefix*)

Return a new model where all field names have been prefixed.

Parameters
prefix (str) – String prefix to add to all field names.

Return type
Type[*Model*]

Returns
New model class with all the same fields only prefixed with the given prefix.

Example

```
>>> class MyModel(Model):
...     a: int
...     b: int
...
```

```
>>> MyModel.prefix("x_").columns
['x_a', 'x_b']
```

patito.Model.rename

classmethod Model.**rename**(*mapping*)

Return a new model class where the specified fields have been renamed.

Parameters
mapping (Dict[str, str]) – A dictionary where the keys are the old field names and the values are the new names.

Return type
Type[*Model*]

Returns
A new model class where the given fields have been renamed.

Raises
ValueError – If non-existent fields are renamed.

Example

```
>>> class MyModel(Model):
...     a: int
...     b: int
...
```

```
>>> MyModel.rename({"a": "A"}).columns
['b', 'A']
```

patito.Model.select

classmethod Model.**select**(*fields*)

Create a new model consisting of only a subset of the model fields.

Parameters
fields (Union[str, Iterable[str]]) – A single field name as a string or a collection of strings.

Return type
Type[*Model*]

Returns
A new model containing only the fields specified by `fields`.

Raises
ValueError – If one or more non-existent fields are selected.

Example

```
>>> class MyModel(Model):
...     a: int
...     b: int
...     c: int
...
```

```
>>> MyModel.select("a").columns
['a']
```

```
>>> sorted(MyModel.select(["b", "c"]).columns)
['b', 'c']
```

patito.Model.suffix

classmethod Model.**suffix**(*suffix*)

Return a new model where all field names have been suffixed.

Parameters
suffix (str) – String suffix to add to all field names.

Return type
Type[*Model*]

Returns
New model class with all the same fields only suffixed with the given suffix.

Example

```
>>> class MyModel(Model):
...     a: int
...     b: int
...
```

```
>>> MyModel.suffix("_x").columns
['a_x', 'b_x']
```

patito.Model.validate

classmethod Model.**validate**(*dataframe*)

Validate the schema and content of the given dataframe.

Parameters

dataframe (Union[DataFrame, DataFrame]) – Polars DataFrame to be validated.

Raises

patito.exceptions.ValidationError – If the given dataframe does not match the given schema.

Examples

```
>>> import patito as pt
>>> import polars as pl
```

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     temperature_zone: Literal["dry", "cold", "frozen"]
...     is_for_sale: bool
...
```

```
>>> df = pl.DataFrame(
...     {
...         "product_id": [1, 1, 3],
...         "temperature_zone": ["dry", "dry", "oven"],
...     }
... )
>>> try:
...     Product.validate(df)
... except pt.ValidationError as exc:
...     print(exc)
...
3 validation errors for Product
is_for_sale
  Missing column (type=type_error.missingcolumns)
product_id
  2 rows with duplicated values. (type=value_error.rowvalue)
temperature_zone
  Rows with invalid values: {'oven'}. (type=value_error.rowvalue)
```

Return type

None

patito.Model.with_fields

classmethod Model.**with_fields**(***field_definitions*)

Return a new model class where the given fields have been added.

Parameters

****field_definitions** (Any) – the keywords are of the form: `field_name=(field_type, field_default)`. Specify `...` if no default value is provided. For instance, `column_name=(int, ...)` will create a new non-optional integer field named `"column_name"`.

Return type

Type[*Model*]

Returns

A new model with all the original fields and the additional field definitions.

Example

```
>>> class MyModel(Model):
...     a: int
...
>>> class ExpandedModel(MyModel):
...     b: int
...
>>> MyModel.with_fields(b=(int, ...)).columns == ExpandedModel.columns
True
```

2.3 patito.Field

class patito.**Field**(*default=PydanticUndefined*, ***, *default_factory=None*, *alias=None*, *title=None*, *description=None*, *exclude=None*, *include=None*, *const=None*, *gt=None*, *ge=None*, *lt=None*, *le=None*, *multiple_of=None*, *allow_inf_nan=None*, *max_digits=None*, *decimal_places=None*, *min_items=None*, *max_items=None*, *unique_items=None*, *min_length=None*, *max_length=None*, *allow_mutation=True*, *regex=None*, *discriminator=None*, *repr=True*, ***extra*)

Annotate model field with additional type and validation information.

This class is built on `pydantic.Field` and you can find its full documentation here. Patito adds additional parameters which are used when validating dataframes, these are documented here.

Parameters

- **constraints** (*Union[polars.Expression, List[polars.Expression]]*) – A single constraint or list of constraints, expressed as a polars expression objects. All rows must satisfy the given constraint. You can refer to the given column with `pt.field`, which will automatically be replaced with `polars.col(<field_name>)` before evaluation.
- **unique** (*bool*) – All row values must be unique.
- **dtype** (*polars.datatype.DataType*) – The given dataframe column must have the given polars dtype, for instance `polars.UInt64` or `pl.Float32`.
- **gt** (*float*) – All values must be greater than `gt`.

- **ge** (*float*) – All values must be greater than or equal to `ge`.
- **lt** (*float*) – All values must be less than `lt`.
- **le** (*float*) – All values must be less than or equal to `lt`.
- **multiple_of** (*float*) – All values must be multiples of the given value.
- **const** (*bool*) – If set to `True` *all* values must be equal to the provided default value, the first argument provided to the `Field` constructor.
- **regex** (*str*) – UTF-8 string column must match regex pattern for all row values.
- **min_length** (*int*) – Minimum length of all string values in a UTF-8 column.
- **max_length** (*int*) – Maximum length of all string values in a UTF-8 column.

Returns
Object used to represent additional constraints put upon the given field.

Return type
FieldInfo

Examples

```
>>> import patito as pt
>>> import polars as pl
>>> class Product(pt.Model):
...     # Do not allow duplicates
...     product_id: int = pt.Field(unique=True)
...
...     # Price must be stored as unsigned 16-bit integers
...     price: int = pt.Field(dtype=pl.UInt16)
...
...     # The product name should be from 3 to 128 characters long
...     name: str = pt.Field(min_length=3, max_length=128)
...
...     # Represent colors in the form of upper cased hex colors
...     brand_color: str = pt.Field(regex=r"^\#[0-9A-F]{6}$")
...
>>> Product.DataFrame(
...     {
...         "product_id": [1, 1],
...         "price": [400, 600],
...         "brand_color": ["#ab00ff", "AB00FF"],
...     }
... ).validate()
Traceback (most recent call last):
  ...
patito.exceptions.ValidationError: 4 validation errors for Product
name
  Missing column (type=type_error.missingcolumns)
product_id
  2 rows with duplicated values. (type=value_error.rowvalue)
price
  Polars dtype <class 'polars.datatypes.Int64'>          does not match model␣
↪field type. (type=type_error.columndtype)
```

(continues on next page)

(continued from previous page)

```
brand_color
  2 rows with out of bound values. (type=value_error.rowvalue)
```

2.4 patito.Database

`Database.__init__`(*path=None*, *read_only=False*, ***kwargs*)

Instantiate a new DuckDB database, either persisted to disk or in-memory.

Parameters

- **path** (`Optional[Path]`) – Optional path to store all the data to. If `None` the data is persisted in-memory only.
- **read_only** (`bool`) – If the database connection should be a read-only connection.
- ****kwargs** (`Any`) – Additional keywords forwarded to `duckdb.connect()`.

Examples

```
>>> import patito as pt
>>> db = pt.Database()
>>> db.to_relation("select 1 as a, 2 as b").create_table("my_table")
>>> db.query("select * from my_table").to_df()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 2   │
└─────┴─────┘
```

2.4.1 Methods

patito.Database.create_enum_types

`Database.create_enum_types`(*model*)

Define SQL enum types in DuckDB database.

Parameters

model (`Type[TypeVar(ModelType, bound= Model)]`) – Model for which all Literal-annotated or enum-annotated string fields will get respective DuckDB enum types.

Example

```
>>> import patito as pt
>>> class EnumModel(pt.Model):
...     enum_column: Literal["A", "B", "C"]
...
>>> db = pt.Database()
>>> db.create_enum_types(EnumModel)
>>> db.enum_types
{'enum__7ba49365cc1b0fd57e61088b3bc9aa25'}
```

Return type
None

patito.Database.create_table

Database.**create_table**(*name*, *model*)

Create table with schema matching the provided Patito model.

See *Relation.insert_into()* for how to insert data into the table after creation. The *Relation.create_table()* method can also be used to create a table from a given relation *and* insert the data at the same time.

Parameters

- **name** (str) – Name of new database table.
- **model** (*Type[Model]*) – Patito model indicating names and types of table columns.

Returns
Relation pointing to the new table.

Return type
Relation[ModelType]

Example

```
>>> from typing import Optional
>>> import patito as pt
>>> class MyModel(pt.Model):
...     str_column: str
...     nullable_string_column: Optional[str]
...
>>> db = pt.Database()
>>> db.create_table(name="my_table", model=MyModel)
>>> db.table("my_table").types
{'str_column': 'VARCHAR', 'nullable_string_column': 'VARCHAR'}
```

patito.Database.create_view

Database.**create_view**(*name*, *data*)

Create a view based on the given data source.

Return type
Relation

patito.Database.default

classmethod Database.**default**()

Return the default DuckDB database.

Return type
Database

Returns
A patito *Database* object wrapping around the given connection.

Example

```
>>> import patito as pt
>>> db = pt.Database.default()
>>> db.query("select 1 as a, 2 as b").to_df()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 2   │
└─────┴─────┘
```

patito.Database.empty_relation

Database.**empty_relation**(*schema*)

Create relation with zero rows, but correct schema that matches the given model.

Parameters
schema (Type[TypeVar(ModelType, bound= Model)]) – A patito model which specifies the column names and types of the given relation.

Example

```
>>> import patito as pt
>>> class Schema(pt.Model):
...     string_column: str
...     bool_column: bool
...
>>> db = pt.Database()
```

(continues on next page)

(continued from previous page)

```
>>> empty_relation = db.empty_relation(Schema)
>>> empty_relation.to_df()
shape: (0, 2)
┌─────────────┬─────────────┐
│ string_column ┆ bool_column │
│ ---         ┆ ---         │
│ str         ┆ bool        │
└─────────────┴─────────────┘
>>> non_empty_relation = db.query(
...     "select 'dummy' as string_column, true as bool_column"
... )
>>> non_empty_relation.union(empty_relation).to_df()
shape: (1, 2)
┌─────────────┬─────────────┐
│ string_column ┆ bool_column │
│ ---         ┆ ---         │
│ str         ┆ bool        │
╞═════════════╪═════════════╡
│ dummy       ┆ true        │
└─────────────┴─────────────┘
```

Return type
Relation[TypeVar(ModelType, bound= Model)]

patito.Database.execute

Database.**execute**(*query*, **parameters*)

Execute SQL query in DuckDB database.

Parameters

- **query** (str) – A SQL statement to execute. Does *not* have to be terminated with a semicolon (;).
- **parameters** (Collection[Union[str, int, float, bool]]) – One or more sets of parameters to insert into prepared statements. The values are replaced in place of the question marks (?) in the prepared query.

Example

```
>>> import patito as pt
>>> db = pt.Database()
>>> db.execute("create table my_table (x bigint);")
>>> db.execute("insert into my_table values (1), (2), (3)")
>>> db.table("my_table").to_df()
shape: (3, 1)
┌─────┐
│ x   │
│ --- │
│ i64 │
```

(continues on next page)

(continued from previous page)

```
┌─────┐
│ 1   │
├╌╌╌╌╌┤
│ 2   │
├╌╌╌╌╌┤
│ 3   │
└─────┘
```

Parameters can be specified when executing prepared queries.

```
>>> db.execute("delete from my_table where x = ?", (2,))
>>> db.table("my_table").to_df()
shape: (2, 1)
┌─────┐
│ x   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
├╌╌╌╌╌┤
│ 3   │
└─────┘
```

Multiple parameter sets can be specified when executing multiple prepared queries.

```
>>> db.execute(
...     "delete from my_table where x = ?",
...     (1,),
...     (3,),
... )
>>> db.table("my_table").to_df()
shape: (0, 1)
┌─────┐
│ x   │
│ --- │
│ i64 │
╞═════╡
└─────┘
```

Return type
None

patito.Database.from_connection

classmethod Database.**from_connection**(*connection*)

Create database from native DuckDB connection object.

Parameters
connection (DuckDBPyConnection) – A native DuckDB connection object created with duckdb.connect().

Return type
Database

Returns

A *Database* object wrapping around the given connection.

Example

```
>>> import duckdb
>>> import patito as pt
>>> connection = duckdb.connect()
>>> database = pt.Database.from_connection(connection)
```

patito.Database.query

`Database.`**`query`**(*query*, *alias='query_relation'*)

Execute arbitrary SQL select query and return the relation.

Parameters

- **query** (str) – Arbitrary SQL select query.
- **alias** (str) – The alias to assign to the resulting relation, to be used in further queries.

Returns: A relation representing the data produced by the given query.

Example

```
>>> import patito as pt
>>> db = pt.Database()
>>> relation = db.query("select 1 as a, 2 as b, 3 as c")
>>> relation.to_df()
shape: (1, 3)
┌─────┬─────┬─────┐
│ a   ┆ b   ┆ c   │
│ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╡
│ 1   ┆ 2   ┆ 3   │
└─────┴─────┴─────┘
```

```
>>> relation = db.query("select 1 as a, 2 as b, 3 as c", alias="my_alias")
>>> relation.select("my_alias.a").to_df()
shape: (1, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
└─────┘
```

Return type

Relation

patito.Database.table

`Database.table(name)`

Return relation representing all the data in the given table.

Parameters

name (`str`) – The name of the table.

Example

```
>>> import patito as pt
>>> df = pt.DataFrame({"a": [1, 2], "b": [3, 4]})
>>> db = pt.Database()
>>> relation = db.to_relation(df)
>>> relation.create_table(name="my_table")
>>> db.table("my_table").to_df()
shape: (2, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 3   │
├╌╌╌╌╌┼╌╌╌╌╌┤
│ 2   ┆ 4   │
└─────┴─────┘
```

Return type

`Relation`

patito.Database.to_relation

`Database.to_relation(derived_from)`

Create a new relation object based on data source.

The given data will be represented as a relation associated with the database. `Database(x).to_relation(y)` is equivalent to `Relation(y, database=Database(x))`.

Parameters

derived_from (*RelationSource*) – One of either a polars or pandas `DataFrame`, a `pathlib.Path` to a parquet or CSV file, a SQL query string, or an existing relation.

Example

```
>>> import patito as pt
>>> db = pt.Database()
>>> db.to_relation("select 1 as a, 2 as b").to_df()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
```

(continues on next page)

(continued from previous page)

```
| i64  i64 |

| 1    2   |
└──────────┘
>>> db.to_relation(pt.DataFrame({"c": [3, 4], "d": ["5", "6"]})).to_df()
shape: (2, 2)
┌─────┬─────┐
| c   | d   |
| --- | --- |
| i64 | str |
╞═════╪═════╡
| 3   | 5   |
├╌╌╌╌╌┼╌╌╌╌╌┤
| 4   | 6   |
└─────┴─────┘
```

Return type
Relation

patito.Database.__contains__

Database.**__contains__**(*table*)

Return True if the database contains a table with the given name.

Parameters
table (str) – The name of the table to be checked for.

Examples

```
>>> import patito as pt
>>> db = pt.Database()
>>> "my_table" in db
False
>>> db.to_relation("select 1 as a, 2 as b").create_table(name="my_table")
>>> "my_table" in db
True
```

Return type
bool

2.5 patito.Relation

`Relation.__init__`(*derived_from*, *database=None*, *model=None*)

Create a new relation object containing data to be queried with DuckDB.

Parameters

- **derived_from** (Union[*DataFrame*, DataFrame, DataFrame, Path, str, DuckDBPyRelation, Relation]) – Data to be represented as a DuckDB relation object. Can be one of the following types:
  - A pandas or polars DataFrame.
  - An SQL query represented as a string.
  - A `Path` object pointing to a CSV or a parquet file. The path must point to an existing file with either a `.csv` or `.parquet` file extension.
  - A native DuckDB relation object (`duckdb.DuckDBPyRelation`).
  - A `patito.Relation` object.
- **database** (Optional[Database]) – Which database to load the relation into. If not provided, the default DuckDB database will be used.
- **model** (Optional[Type[TypeVar(ModelType, bound= Model)]]) – Sub-class of `patito.Model` which specifies how to deserialize rows when fetched with methods such as *Relation.get()* and `__iter__()`.

  Will also be used to create a strict table schema if *Relation.create_table()*. schema should be constructed.

  If not provided, a dynamic model fitting the relation schema will be created when required.

  Can also be set later dynamically by invoking *Relation.set_model()*.

Raises

- **ValueError** – If any one of the following cases are encountered:

  - If a provided `Path` object does not have a `.csv` or `.parquet` file extension. - If a database and relation object is provided, but the relation object does not belong to the database.
- **TypeError** – If the type of `derived_from` is not supported.

Examples

Instantiated from a dataframe:

```
>>> import patito as pt
>>> df = pt.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
>>> pt.Relation(df).filter("a > 2").to_df()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 3   ┆ 6   │
└─────┴─────┘
```

Instantiated from an SQL query:

```
>>> pt.Relation("select 1 as a, 2 as b").to_df()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 2   │
└─────┴─────┘
```

2.5.1 Properties

patito.Relation.alias

The alias that can be used to refer to the given relation in queries. Can be set with *Relation.set_alias()*.

patito.Relation.columns

property Relation.**columns: List[str]**

Return the columns of the relation as a list of strings.

Examples

```
>>> import patito as pt
>>> pt.Relation("select 1 as a, 2 as b").columns
['a', 'b']
```

Return type
List[str]

patito.Relation.model

The model associated with the relation, if set with *Relation.set_model()*.

patito.Relation.types

Relation.**types**

Return the SQL types of all the columns of the given relation.

Returns
A dictionary where the keys are the column names and the values are SQL types as strings.

Return type
dict[str, str]

Examples

```
>>> import patito as pt
>>> pt.Relation("select 1 as a, 'my_value' as b").types
{'a': 'INTEGER', 'b': 'VARCHAR'}
```

2.5.2 Methods

patito.Relation.add_prefix

`Relation.`**`add_prefix`**(*prefix*, *include=None*, *exclude=None*)

Add a prefix to all the columns of the relation.

Parameters

- **prefix** (`str`) – A string to prepend to add to all the columns names.
- **include** (`Optional[Iterable[str]]`) – If provided, only the given columns will be renamed.
- **exclude** (`Optional[Iterable[str]]`) – If provided, the given columns will *not* be renamed.

Raises

TypeError – If both include *and* exclude are provided at the same time.

Examples

```
>>> import patito as pt
>>> relation = pt.Relation("select 1 as column_1, 2 as column_2")
>>> relation.add_prefix("renamed_").to_df()
shape: (1, 2)
┌──────────────────┬──────────────────┐
│ renamed_column_1 ┆ renamed_column_2 │
│ ---              ┆ ---              │
│ i64              ┆ i64              │
╞══════════════════╪══════════════════╡
│ 1                ┆ 2                │
└──────────────────┴──────────────────┘
```

```
>>> relation.add_prefix("renamed_", include=["column_1"]).to_df()
shape: (1, 2)
┌──────────────────┬──────────┐
│ renamed_column_1 ┆ column_2 │
│ ---              ┆ ---      │
│ i64              ┆ i64      │
╞══════════════════╪══════════╡
│ 1                ┆ 2        │
└──────────────────┴──────────┘
```

```
>>> relation.add_prefix("renamed_", exclude=["column_1"]).to_df()
shape: (1, 2)
```

(continues on next page)

(continued from previous page)

```
┌──────────┬──────────────────┐
│ column_1 ┆ renamed_column_2 │
│ ---      ┆ ---              │
│ i64      ┆ i64              │
╞══════════╪══════════════════╡
│ 1        ┆ 2                │
└──────────┴──────────────────┘
```

Return type
Relation

patito.Relation.add_suffix

Relation.**add_suffix**(*suffix*, *include=None*, *exclude=None*)

Add a suffix to all the columns of the relation.

Parameters

- **suffix** (str) – A string to append to add to all columns names.
- **include** (Optional[Collection[str]]) – If provided, only the given columns will be renamed.
- **exclude** (Optional[Collection[str]]) – If provided, the given columns will *not* be renamed.

Raises
TypeError – If both include *and* exclude are provided at the same time.

Examples

```
>>> import patito as pt
>>> relation = pt.Relation("select 1 as column_1, 2 as column_2")
>>> relation.add_suffix("_renamed").to_df()
shape: (1, 2)
┌────────────────┬────────────────┐
│ column_1_renamed ┆ column_2_renamed │
│ ---            ┆ ---            │
│ i64            ┆ i64            │
╞════════════════╪════════════════╡
│ 1              ┆ 2              │
└────────────────┴────────────────┘
```

```
>>> relation.add_suffix("_renamed", include=["column_1"]).to_df()
shape: (1, 2)
┌────────────────┬──────────┐
│ column_1_renamed ┆ column_2 │
│ ---            ┆ ---      │
│ i64            ┆ i64      │
╞════════════════╪══════════╡
│ 1              ┆ 2        │
└────────────────┴──────────┘
```

```
>>> relation.add_suffix("_renamed", exclude=["column_1"]).to_df()
shape: (1, 2)
┌──────────┬────────────────┐
│ column_1 ┆ column_2_renamed │
│ ---      ┆ ---            │
│ i64      ┆ i64            │
╞══════════╪════════════════╡
│ 1        ┆ 2              │
└──────────┴────────────────┘
```

Return type
Relation

patito.Relation.aggregate

Relation.**aggregate**(**aggregations*, *group_by*, ***named_aggregations*)

Return relation formed by GROUP BY SQL aggregation(s).

Parameters

- **aggregations** (str) – Zero or more aggregation expressions such as "sum(column_name)" and "count(distinct column_name)".
- **named_aggregations** (str) – Zero or more aggregated expressions where the keyword is used to name the given aggregation. For example, my_column="sum(column_name)" is inserted as "sum(column_name) as my_column" in the executed SQL query.
- **group_by** (Union[str, Iterable[str]]) – A single column name or iterable collection of column names to group by.

Examples

```
>>> import patito as pt
>>> df = pt.DataFrame({"a": [1, 2, 3], "b": ["X", "Y", "X"]})
>>> relation = pt.Relation(df)
>>> relation.aggregate(
...     "b",
...     "sum(a)",
...     "greatest(b)",
...     max_a="max(a)",
...     group_by="b",
... ).to_df()
shape: (2, 4)
┌─────┬────────┬─────────────┬───────┐
│ b   ┆ sum(a) ┆ greatest(b) ┆ max_a │
│ --- ┆ ---    ┆ ---         ┆ ---   │
│ str ┆ f64    ┆ str         ┆ i64   │
╞═════╪════════╪═════════════╪═══════╡
│ X   ┆ 4.0    ┆ X           ┆ 3     │
├╌╌╌╌╌┼╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌┤
│ Y   ┆ 2.0    ┆ Y           ┆ 2     │
└─────┴────────┴─────────────┴───────┘
```

Return type
Relation

patito.Relation.all

Relation.**all**(**filters*, ***equalities*)

Return True if the given predicate(s) are true for all rows in the relation.

See *Relation.filter()* for additional information regarding the parameters.

Parameters

- **filters** (str) – SQL predicates to satisfy.
- **equalities** (Union[int, float, str]) – SQL equality predicates to satisfy.

Examples

```
>>> import patito as pt
>>> df = pt.DataFrame(
...     {
...         "even_number": [2, 4, 6],
...         "odd_number": [1, 3, 5],
...         "zero": [0, 0, 0],
...     }
... )
>>> relation = pt.Relation(df)
>>> relation.all(zero=0)
True
>>> relation.all(
...     "even_number % 2 = 0",
...     "odd_number % 2 = 1",
...     zero=0,
... )
True
>>> relation.all(zero=1)
False
>>> relation.all("odd_number % 2 = 0")
False
```

Return type
bool

patito.Relation.case

Relation.**case**(*, *from_column*, *to_column*, *mapping*, *default*)

Map values of one column over to a new column.

Parameters

- **from_column** (str) – Name of column defining the domain of the mapping.
- **to_column** (str) – Name of column to insert the mapped values into.

- **mapping** (Dict[Union[str, float, int, None], Union[str, float, int, None]]) – Dictionary defining the mapping. The dictionary keys represent the input values, while the dictionary values represent the output values. Items are inserted into the SQL case statement by their repr() string value.
- **default** (Union[str, float, int, None]) – Default output value for inputs which have no provided mapping.

Examples

The following case statement...

```
>>> import patito as pt
>>> db = pt.Database()
>>> relation = db.to_relation("select 1 as a union select 2 as a")
>>> relation.case(
...     from_column="a",
...     to_column="b",
...     mapping={1: "one", 2: "two"},
...     default="three",
... ).order(by="a").to_df()
shape: (2, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ str │
╞═════╪═════╡
│ 1   ┆ one │
├╌╌╌╌╌┼╌╌╌╌╌┤
│ 2   ┆ two │
└─────┴─────┘
```

... is equivalent with:

```
>>> case_statement = pt.sql.Case(
...     on_column="a",
...     mapping={1: "one", 2: "two"},
...     default="three",
...     as_column="b",
... )
>>> relation.select(f"*, {case_statement}").order(by="a").to_df()
shape: (2, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ str │
╞═════╪═════╡
│ 1   ┆ one │
├╌╌╌╌╌┼╌╌╌╌╌┤
│ 2   ┆ two │
└─────┴─────┘
```

Return type
Relation

patito.Relation.cast

`Relation.cast`(*model=None*, *strict=False*, *include=None*, *exclude=None*)

Cast the columns of the relation to types compatible with the associated model.

The associated model must either be set by invoking *Relation.set_model()* or provided with the `model` parameter.

Any columns of the relation that are not part of the given model schema will be left as-is.

Parameters

- **model** (Optional[TypeVar(ModelType, bound= Model)]) – If *Relation.set_model()* has not been invoked or is intended to be overwritten.
- **strict** (bool) – If set to `False`, columns which are technically compliant with the specified field type, will not be casted. For example, a column annotated with `int` is technically compliant with `SMALLINT`, even if `INTEGER` is the default SQL type associated with `int`-annotated fields. If `strict` is set to `True`, the resulting dtypes will be forced to the default dtype associated with each python type.
- **include** (Optional[Collection[str]]) – If provided, only the given columns will be casted.
- **exclude** (Optional[Collection[str]]) – If provided, the given columns will *not* be casted.

Return type

TypeVar(RelationType, bound= Relation)

Returns

New relation where the columns have been casted according to the model schema.

Examples

```
>>> import patito as pt
>>> class Schema(pt.Model):
...     float_column: float
...
>>> relation = pt.Relation("select 1 as float_column")
>>> relation.types["float_column"]
'INTEGER'
>>> relation.cast(model=Schema).types["float_column"]
'DOUBLE'
```

```
>>> relation = pt.Relation("select 1::FLOAT as float_column")
>>> relation.cast(model=Schema).types["float_column"]
'FLOAT'
>>> relation.cast(model=Schema, strict=True).types["float_column"]
'DOUBLE'
```

```
>>> class Schema(pt.Model):
...     column_1: float
...     column_2: float
...
>>> relation = pt.Relation("select 1 as column_1, 2 as column_2").set_model(
...     Schema
```

(continues on next page)

(continued from previous page)

```
... )
>>> relation.types
{'column_1': 'INTEGER', 'column_2': 'INTEGER'}
>>> relation.cast(include=["column_1"]).types
{'column_1': 'DOUBLE', 'column_2': 'INTEGER'}
>>> relation.cast(exclude=["column_1"]).types
{'column_1': 'INTEGER', 'column_2': 'DOUBLE'}
```

patito.Relation.coalesce

Relation.**coalesce**(***column_expressions*)

Replace null-values in given columns with respective values.

For example, `coalesce(column_name=value)` is compiled to: `f"coalesce({column_name}, {repr(value)}) as column_name"` in the resulting SQL.

Parameters

column_expressions (Union[str, int, float]) – Keywords indicate which columns to coalesce, while the string representation of the respective arguments are used as the null-replacement.

Returns

Relation where values have been filled in for nulls in the given columns.

Return type

Relation

Examples

```
>>> import patito as pt
>>> df = pt.DataFrame(
...     {
...         "a": [1, None, 3],
...         "b": ["four", "five", None],
...         "c": [None, 8.0, 9.0],
...     }
... )
>>> relation = pt.Relation(df)
>>> relation.coalesce(a=2, b="six").to_df()
shape: (3, 3)
┌─────┬──────┬──────┐
│ a   ┆ b    ┆ c    │
│ --- ┆ ---  ┆ ---  │
│ i64 ┆ str  ┆ f64  │
╞═════╪══════╪══════╡
│ 1   ┆ four ┆ null │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 2   ┆ five ┆ 8.0  │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 3   ┆ six  ┆ 9.0  │
└─────┴──────┴──────┘
```

patito.Relation.count

`Relation.`**`count`**`()`

Return the number of rows in the given relation.

Return type
`int`

Returns
Number of rows in the relation as an integer.

Examples

```
>>> import patito as pt
>>> relation = pt.Relation("select 1 as a")
>>> relation.count()
1
>>> (relation + relation).count()
2
```

The *Relation.__len__()* method invokes `Relation.count()` under the hood, and is equivalent:

```
>>> len(relation)
1
>>> len(relation + relation)
2
```

patito.Relation.create_table

`Relation.`**`create_table`**`(`*`name`*`)`

Create new database table based on relation.

If `self.model` is set with *Relation.set_model()*, then the model is used to infer the table schema. Otherwise, a permissive table schema is created based on the relation data.

Returns
A relation pointing to the newly created table.

Return type
Relation

Examples

```
>>> from typing import Literal
>>> import patito as pt
```

```
>>> df = pt.DataFrame({"enum_column": ["A", "A", "B"]})
>>> relation = pt.Relation(df)
>>> relation.create_table("permissive_table").types
{'enum_column': 'VARCHAR'}
```

```
>>> class TableSchema(pt.Model):
...     enum_column: Literal["A", "B", "C"]
...
>>> relation.set_model(TableSchema).create_table("strict_table").types
{'enum_column': 'enum__7ba49365cc1b0fd57e61088b3bc9aa25'}
```

patito.Relation.create_view

`Relation.`**`create_view`**(*name*, *replace=False*)

Create new database view based on relation.

Returns
A relation pointing to the newly created view.

Return type
Relation

Examples

```
>>> import patito as pt
>>> db = pt.Database()
>>> df = pt.DataFrame({"column": ["A", "A", "B"]})
>>> relation = db.to_relation(df)
>>> relation.create_view("my_view")
>>> db.query("select * from my_view").to_df()
shape: (3, 1)
┌────────┐
│ column │
│ ---    │
│ str    │
╞════════╡
│ A      │
├╌╌╌╌╌╌╌╌┤
│ A      │
├╌╌╌╌╌╌╌╌┤
│ B      │
└────────┘
```

patito.Relation.distinct

`Relation.`**`distinct`**()

Drop all duplicate rows of the relation.

Example

```
>>> import patito as pt
>>> df = pt.DataFrame(
...     [[1, 2, 3], [1, 2, 3], [3, 2, 1]],
...     columns=["a", "b", "c"],
...     orient="row",
... )
>>> relation = pt.Relation(df)
>>> relation.to_df()
shape: (3, 3)
┌─────┬─────┬─────┐
│ a   ┆ b   ┆ c   │
│ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╡
│ 1   ┆ 2   ┆ 3   │
├╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌┤
│ 1   ┆ 2   ┆ 3   │
├╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌┤
│ 3   ┆ 2   ┆ 1   │
└─────┴─────┴─────┘
>>> relation.distinct().to_df()
shape: (2, 3)
┌─────┬─────┬─────┐
│ a   ┆ b   ┆ c   │
│ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╡
│ 1   ┆ 2   ┆ 3   │
├╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌┤
│ 3   ┆ 2   ┆ 1   │
└─────┴─────┴─────┘
```

Return type
TypeVar(RelationType, bound= Relation)

patito.Relation.drop

Relation.**drop**(**columns*)

Remove specified column(s) from relation.

Parameters
columns (*str*) – Any number of string column names to be dropped.

Examples

```
>>> import patito as pt
>>> relation = pt.Relation("select 1 as a, 2 as b, 3 as c")
>>> relation.columns
['a', 'b', 'c']
>>> relation.drop("c").columns
['a', 'b']
>>> relation.drop("b", "c").columns
['a']
```

Return type
Relation

patito.Relation.except_

Relation.**except_**(*other*)

Remove all rows that can be found in the other other relation.

Parameters
other (Union[*DataFrame*, DataFrame, DataFrame, Path, str, DuckDBPyRelation, Relation]) – Another relation or something that can be casted to a relation.

Return type
TypeVar(RelationType, bound= Relation)

Returns
New relation without the rows that can be found in the other relation.

Example

```
>>> import patito as pt
>>> relation_123 = pt.Relation("select 1 union select 2 union select 3")
>>> relation_123.order(by="1").to_df()
shape: (3, 1)
┌─────┐
│ 1   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
├╌╌╌╌╌┤
│ 2   │
├╌╌╌╌╌┤
│ 3   │
└─────┘
>>> relation_2 = pt.Relation("select 2")
>>> relation_2.to_df()
shape: (1, 1)
┌─────┐
│ 2   │
│ --- │
```

(continues on next page)

(continued from previous page)

```
│ i64 │
│ 2   │
└─────┘
>>> relation_123.except_(relation_2).order(by="1").to_df()
shape: (2, 1)
┌─────┐
│ 1   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
├╌╌╌╌╌┤
│ 3   │
└─────┘
```

patito.Relation.execute

`Relation.`**`execute`**`()`

Execute built relation query and return result object.

Return type
`DuckDBPyResult`

Returns
A native `duckdb.DuckDBPyResult` object representing the executed query.

Examples

```
>>> import patito as pt
>>> relation = pt.Relation(
...     "select 1 as a, 2 as b union select 3 as a, 4 as b"
... )
>>> result = relation.aggregate("sum(a)", group_by="").execute()
>>> result.description()
[('sum(a)', 'NUMBER', None, None, None, None, None)]
>>> result.fetchall()
[(4,)]
```

patito.Relation.filter

`Relation.`**`filter`**`(*filters, **equalities)`

Return subset of rows of relation that satisfy the given predicates.

The method returns self if no filters are provided.

Parameters

- **`filters`** (`str`) – A conjunction of SQL `WHERE` clauses.
- **`equalities`** (`Union[str, int, float]`) – A conjunction of SQL equality clauses. The keyword name is the column and the parameter is the value of the equality.

Returns
A new relation where all rows satisfy the given criteria.

Return type
Relation

Examples

```
>>> import patito as pt
>>> df = pt.DataFrame(
...     {
...         "number": [1, 2, 3, 4],
...         "string": ["A", "A", "B", "B"],
...     }
... )
>>> relation = pt.Relation(df)
>>> relation.filter("number % 2 = 0").to_df()
shape: (2, 2)
┌────────┬────────┐
│ number ┆ string │
│ ---    ┆ ---    │
│ i64    ┆ str    │
╞════════╪════════╡
│ 2      ┆ A      │
├╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌┤
│ 4      ┆ B      │
└────────┴────────┘
```

```
>>> relation.filter(number=1, string="A").to_df()
shape: (1, 2)
┌────────┬────────┐
│ number ┆ string │
│ ---    ┆ ---    │
│ i64    ┆ str    │
╞════════╪════════╡
│ 1      ┆ A      │
└────────┴────────┘
```

patito.Relation.get

`Relation.get(*filters, **equalities)`

Fetch the single row that matches the given filter(s).

If you expect a relation to already return one row, you can use get() without any arguments to return that row.

Raises
RuntimeError – RuntimeError is thrown if not exactly one single row matches the given filter.

Parameters

- **filters** (*str*) – A conjunction of SQL where clauses.
- **equalities** (*Any*) – A conjunction of SQL equality clauses. The keyword name is the column and the parameter is the value of the equality.

Returns
A Patito model representing the given row.

Return type
Model

Examples

```
>>> import patito as pt
>>> import polars as pl
>>> df = pt.DataFrame({"product_id": [1, 2, 3], "price": [10, 10, 20]})
>>> relation = pt.Relation(df).set_alias("my_relation")
```

The `.get()` method will by default return a dynamically constructed Patito model if no model has been associated with the given relation:

```
>>> relation.get(product_id=1)
my_relation(product_id=1, price=10)
```

If a Patito model has been associated with the relation, by the use of *Relation.set_model()*, then the given model will be used to represent the return type:

```
>>> class Product(pt.Model):
...     product_id: int = pt.Field(unique=True)
...     price: float
...
>>> relation.set_model(Product).get(product_id=1)
Product(product_id=1, price=10.0)
```

You can invoke `.get()` without any arguments on relations containing exactly one row:

```
>>> relation.filter(product_id=1).get()
my_relation(product_id=1, price=10)
```

If the given predicate matches multiple rows a `MultipleRowsReturned` exception will be raised:

```
>>> try:
...     relation.get(price=10)
... except pt.exceptions.MultipleRowsReturned as e:
...     print(e)
...
Relation.get(price=10) returned 2 rows!
```

If the given predicate matches zero rows a `RowDoesNotExist` exception will be raised:

```
>>> try:
...     relation.get(price=0)
... except pt.exceptions.RowDoesNotExist as e:
...     print(e)
...
Relation.get(price=0) returned 0 rows!
```

patito.Relation.inner_join

`Relation.`**`inner_join`**(*other*, *on*)

Inner join relation with other relation source based on condition.

Parameters

- **other** (Union[*DataFrame*, DataFrame, DataFrame, Path, str, DuckDBPyRelation, Relation]) – A source which can be casted to a `Relation` object, and be used as the right table in the join.
- **on** (str) – Join condition following the `INNER JOIN ... ON` in the SQL query.

Returns

New relation based on the joined relations.

Return type

Relation

Example

```
>>> import patito as pt
>>> products_df = pt.DataFrame(
...     {
...         "product_name": ["apple", "banana", "oranges"],
...         "supplier_id": [2, 1, 3],
...     }
... )
>>> products = pt.Relation(products_df)
>>> supplier_df = pt.DataFrame(
...     {
...         "id": [1, 2],
...         "supplier_name": ["Banana Republic", "Applies Inc."],
...     }
... )
>>> suppliers = pt.Relation(supplier_df)
>>> products.set_alias("p").inner_join(
...     suppliers.set_alias("s"),
...     on="p.supplier_id = s.id",
... ).to_df()
shape: (2, 4)
┌──────────────┬─────────────┬─────┬─────────────────┐
│ product_name ┆ supplier_id ┆ id  ┆ supplier_name   │
│ ---          ┆ ---         ┆ --- ┆ ---             │
│ str          ┆ i64         ┆ i64 ┆ str             │
╞══════════════╪═════════════╪═════╪═════════════════╡
│ apple        ┆ 2           ┆ 2   ┆ Applies Inc.    │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ banana       ┆ 1           ┆ 1   ┆ Banana Republic │
└──────────────┴─────────────┴─────┴─────────────────┘
```

patito.Relation.insert_into

`Relation.`**`insert_into`**(*table*)

Insert all rows of the relation into a given table.

The relation must contain all the columns present in the target table. Extra columns are ignored and the column order is automatically matched with the target table.

Parameters
`table` (`str`) – Name of table for which to insert values into.

Returns
The original relation, i.e. `self`.

Return type
Relation

Examples

```
>>> import patito as pt
>>> db = pt.Database()
>>> db.to_relation("select 1 as a").create_table("my_table")
>>> db.table("my_table").to_df()
shape: (1, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
└─────┘
>>> db.to_relation("select 2 as a").insert_into("my_table")
>>> db.table("my_table").to_df()
shape: (2, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
├╌╌╌╌╌┤
│ 2   │
└─────┘
```

patito.Relation.intersect

`Relation.`**`intersect`**(*other*)

Return a new relation containing the rows that are present in both relations.

This is a set operation which will remove duplicate rows as well.

Parameters

other (Union[*DataFrame*, DataFrame, DataFrame, Path, str, DuckDBPyRelation, Relation]) – Another relation with the same column names.

Returns

A new relation with only those rows that are present in both relations.

Return type

Relation[*Model*]

Example

```
>>> import patito as pt
>>> df1 = pt.DataFrame({"a": [1, 1, 2], "b": [1, 1, 2]})
>>> df2 = pt.DataFrame({"a": [1, 1, 3], "b": [1, 1, 3]})
>>> pt.Relation(df1).intersect(pt.Relation(df2)).to_df()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 1   │
└─────┴─────┘
```

patito.Relation.join

`Relation.`**`join`**(*other*, *, *on*, *how='inner'*)

Join relation with other relation source based on condition.

See *Relation.inner_join()* and *Relation.left_join()* for alternative method shortcuts instead of using `how`.

Parameters

- **other** (Union[*DataFrame*, DataFrame, DataFrame, Path, str, DuckDBPyRelation, Relation]) – A source which can be casted to a `Relation` object, and be used as the right table in the join.
- **on** (str) – Join condition following the `INNER JOIN ... ON` in the SQL query.
- **how** (Literal['inner', 'left']) – Either `"left"` or `"inner"` for what type of SQL join operation to perform.

Returns

New relation based on the joined relations.

Return type

Relation

Example

```
>>> import patito as pt
>>> products_df = pt.DataFrame(
...     {
...         "product_name": ["apple", "banana", "oranges"],
...         "supplier_id": [2, 1, 3],
...     }
... )
>>> products = pt.Relation(products_df)
>>> supplier_df = pt.DataFrame(
...     {
...         "id": [1, 2],
...         "supplier_name": ["Banana Republic", "Applies Inc."],
...     }
... )
>>> suppliers = pt.Relation(supplier_df)
>>> products.set_alias("p").join(
...     suppliers.set_alias("s"),
...     on="p.supplier_id = s.id",
...     how="inner",
... ).to_df()
shape: (2, 4)
┌──────────────┬─────────────┬─────┬─────────────────┐
│ product_name ┆ supplier_id ┆ id  ┆ supplier_name   │
│ ---          ┆ ---         ┆ --- ┆ ---             │
│ str          ┆ i64         ┆ i64 ┆ str             │
╞══════════════╪═════════════╪═════╪═════════════════╡
│ apple        ┆ 2           ┆ 2   ┆ Applies Inc.    │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ banana       ┆ 1           ┆ 1   ┆ Banana Republic │
└──────────────┴─────────────┴─────┴─────────────────┘
```

```
>>> products.set_alias("p").join(
...     suppliers.set_alias("s"),
...     on="p.supplier_id = s.id",
...     how="left",
... ).to_df()
shape: (3, 4)
┌──────────────┬─────────────┬──────┬─────────────────┐
│ product_name ┆ supplier_id ┆ id   ┆ supplier_name   │
│ ---          ┆ ---         ┆ ---  ┆ ---             │
│ str          ┆ i64         ┆ i64  ┆ str             │
╞══════════════╪═════════════╪══════╪═════════════════╡
│ apple        ┆ 2           ┆ 2    ┆ Applies Inc.    │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ banana       ┆ 1           ┆ 1    ┆ Banana Republic │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ oranges      ┆ 3           ┆ null ┆ null            │
└──────────────┴─────────────┴──────┴─────────────────┘
```

patito.Relation.left_join

`Relation.`**`left_join`**(*other*, *on*)

Left join relation with other relation source based on condition.

Parameters

- **other** (Union[*DataFrame*, DataFrame, DataFrame, Path, str, DuckDBPyRelation, Relation]) – A source which can be casted to a Relation object, and be used as the right table in the join.
- **on** (str) – Join condition following the `LEFT JOIN ... ON` in the SQL query.

Returns

New relation based on the joined tables.

Return type

Relation

Example

```
>>> import patito as pt
>>> products_df = pt.DataFrame(
...     {
...         "product_name": ["apple", "banana", "oranges"],
...         "supplier_id": [2, 1, 3],
...     }
... )
>>> products = pt.Relation(products_df)
>>> supplier_df = pt.DataFrame(
...     {
...         "id": [1, 2],
...         "supplier_name": ["Banana Republic", "Applies Inc."],
...     }
... )
>>> suppliers = pt.Relation(supplier_df)
>>> products.set_alias("p").left_join(
...     suppliers.set_alias("s"),
...     on="p.supplier_id = s.id",
... ).to_df()
shape: (3, 4)
┌──────────────┬─────────────┬──────┬─────────────────┐
│ product_name ┆ supplier_id ┆ id   ┆ supplier_name   │
│ ---          ┆ ---         ┆ ---  ┆ ---             │
│ str          ┆ i64         ┆ i64  ┆ str             │
╞══════════════╪═════════════╪══════╪═════════════════╡
│ apple        ┆ 2           ┆ 2    ┆ Applies Inc.    │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ banana       ┆ 1           ┆ 1    ┆ Banana Republic │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ oranges      ┆ 3           ┆ null ┆ null            │
└──────────────┴─────────────┴──────┴─────────────────┘
```

patito.Relation.limit

`Relation.`**`limit`**(*n*, *, *offset=0*)

Remove all but the first n rows.

Parameters

- **n** (int) – The number of rows to keep.
- **offset** (int) – Disregard the first `offset` rows before starting to count which rows to keep.

Return type

TypeVar(RelationType, bound= Relation)

Returns

New relation with only n rows.

Example

```
>>> import patito as pt
>>> relation = (
...     pt.Relation("select 1 as column")
...     + pt.Relation("select 2 as column")
...     + pt.Relation("select 3 as column")
...     + pt.Relation("select 4 as column")
... )
>>> relation.limit(2).to_df()
shape: (2, 1)
┌────────┐
│ column │
│ ---    │
│ i64    │
╞════════╡
│ 1      │
├╌╌╌╌╌╌╌╌┤
│ 2      │
└────────┘
>>> relation.limit(2, offset=2).to_df()
shape: (2, 1)
┌────────┐
│ column │
│ ---    │
│ i64    │
╞════════╡
│ 3      │
├╌╌╌╌╌╌╌╌┤
│ 4      │
└────────┘
```

patito.Relation.order

`Relation.`**order**(*by*)

Change the order of the rows of the relation.

Parameters

by (`Union[str, Iterable[str]]`) – An `ORDER BY` SQL expression such as `"age DESC"` or `("age DESC", "name ASC")`.

Return type

`TypeVar(RelationType, bound= Relation)`

Returns

New relation where the rows have been ordered according to `by`.

Example

```
>>> import patito as pt
>>> df = pt.DataFrame(
...     {
...         "name": ["Alice", "Bob", "Charles", "Diana"],
...         "age": [20, 20, 30, 35],
...     }
... )
>>> df
shape: (4, 2)
┌─────────┬─────┐
│ name    ┆ age │
│ ---     ┆ --- │
│ str     ┆ i64 │
╞═════════╪═════╡
│ Alice   ┆ 20  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Bob     ┆ 20  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Charles ┆ 30  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Diana   ┆ 35  │
└─────────┴─────┘
>>> relation = pt.Relation(df)
>>> relation.order(by="age desc").to_df()
shape: (4, 2)
┌─────────┬─────┐
│ name    ┆ age │
│ ---     ┆ --- │
│ str     ┆ i64 │
╞═════════╪═════╡
│ Diana   ┆ 35  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Charles ┆ 30  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Alice   ┆ 20  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
```

(continues on next page)

(continued from previous page)

```
│ Bob      20  │
└──────────────┘
>>> relation.order(by=["age desc", "name desc"]).to_df()
shape: (4, 2)
┌─────────┬─────┐
│ name    ┆ age │
│ ---     ┆ --- │
│ str     ┆ i64 │
╞═════════╪═════╡
│ Diana   ┆ 35  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Charles ┆ 30  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Bob     ┆ 20  │
├╌╌╌╌╌╌╌╌╌┼╌╌╌╌╌┤
│ Alice   ┆ 20  │
└─────────┴─────┘
```

patito.Relation.select

`Relation.`**`select`**`(`**projections*, ***named_projections*`)`

Return relation based on one or more SQL `SELECT` projections.

Keyword arguments are converted into `{arg} as {keyword}` in the executed SQL query.

Parameters

- ***projections** (`Union[str, int, float]`) – One or more strings representing SQL statements to be selected. For example `"2"` or `"another_column"`.
- ****named_projections** (`Union[str, int, float]`) – One ore more keyword arguments where the keyword specifies the name of the new column and the value is an SQL statement defining the content of the new column. For example `new_column="2 * another_column"`.

Examples

```
>>> import patito as pt
>>> db = pt.Database()
>>> relation = db.to_relation(pt.DataFrame({"original_column": [1, 2, 3]}))
>>> relation.select("*").to_df()
shape: (3, 1)
┌─────────────────┐
│ original_column │
│ ---             │
│ i64             │
╞═════════════════╡
│ 1               │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 2               │
├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
```

(continues on next page)

(continued from previous page)

```
┌──────────────────┐
│ 3                │
└──────────────────┘
>>> relation.select("*", multiplied_column="2 * original_column").to_df()
shape: (3, 2)
┌─────────────────────────────────────┐
│ original_column  multiplied_column  │
│ ---              ---                │
│ i64              i64                │
│                                     │
│ 1                2                  │
├                                     ┤
│ 2                4                  │
├                                     ┤
│ 3                6                  │
└─────────────────────────────────────┘
```

Return type
Relation

patito.Relation.rename

Relation.**rename**(***columns*)

Rename columns as specified.

Parameters
****columns** (str) – A set of keyword arguments where the keyword is the old column name and the value is the new column name.

Raises
ValueError – If any of the given keywords do not exist as columns in the relation.

Examples

```
>>> import patito as pt
>>> relation = pt.Relation("select 1 as a, 2 as b")
>>> relation.rename(b="c").to_df().select(["a", "c"])
shape: (1, 2)
┌──────────┐
│ a    c   │
│ ---  ---  │
│ i64  i64  │
│           │
│ 1    2    │
└──────────┘
```

Return type
Relation

patito.Relation.set_alias

`Relation.`**`set_alias`**(*name*)

Set SQL alias for the given relation to be used in further queries.

Parameters
name (str) – The new alias for the given relation.

Returns
A new relation containing the same query but addressable with the new alias.

Return type
Relation

Example

```
>>> import patito as pt
>>> relation_1 = pt.Relation("select 1 as a, 2 as b")
>>> relation_2 = pt.Relation("select 1 as a, 3 as c")
>>> relation_1.set_alias("x").inner_join(
...     relation_2.set_alias("y"),
...     on="x.a = y.a",
... ).select("x.a", "y.a", "b", "c").to_df()
shape: (1, 4)
┌─────┬─────┬─────┬─────┐
│ a   ┆ a:1 ┆ b   ┆ c   │
│ --- ┆ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╪═════╡
│ 1   ┆ 1   ┆ 2   ┆ 3   │
└─────┴─────┴─────┴─────┘
```

patito.Relation.set_model

`Relation.`**`set_model`**(*model*)

Associate a give Patito model with the relation.

The returned relation has an associated `.model` attribute which can in turn be used by several methods such as *Relation.get()*, *Relation.create_table()*, and *Relation.__iter__*.

Parameters
model – A Patito Model class specifying the intended schema of the relation.

Returns
A new relation with the associated model.

Return type
Relation[model]

Example

```
>>> from typing import Literal
>>> import patito as pt
>>> class MySchema(pt.Model):
...     float_column: float
...     enum_column: Literal["A", "B", "C"]
...
>>> relation = pt.Relation("select 1 as float_column, 'A' as enum_column")
>>> relation.get()
query_relation(float_column=1, enum_column='A')
>>> relation.set_model(MySchema).get()
MySchema(float_column=1.0, enum_column='A')
>>> relation.create_table("unmodeled_table").types
{'float_column': 'INTEGER', 'enum_column': 'VARCHAR'}
>>> relation.set_model(MySchema).create_table("modeled_table").types
{'float_column': 'DOUBLE',
 'enum_column': 'enum__7ba49365cc1b0fd57e61088b3bc9aa25'}
```

patito.Relation.to_df

Relation.**to_df**()

Return a polars DataFrame representation of relation object.

Returns: A `patito.DataFrame` object which inherits from `polars.DataFrame`.

Example

```
>>> import patito as pt
>>> pt.Relation("select 1 as column union select 2 as column").order(
...     by="1"
... ).to_df()
shape: (2, 1)
┌────────┐
│ column │
│ ---    │
│ i64    │
╞════════╡
│ 1      │
├╌╌╌╌╌╌╌╌┤
│ 2      │
└────────┘
```

Return type
DataFrame

patito.Relation.to_pandas

`Relation.to_pandas()`

Return a pandas DataFrame representation of relation object.

Returns: A `pandas.DataFrame` object containing all the data of the relation.

Example

```
>>> import patito as pt
>>> pt.Relation("select 1 as column union select 2 as column").order(
...     by="1"
... ).to_pandas()
      column
   0       1
   1       2
```

Return type
DataFrame

patito.Relation.to_series

`Relation.to_series()`

Convert the given relation to a polars Series.

Raises
TypeError – If the given relation does not contain exactly one column.

Returns: A `polars.Series` object containing the data of the relation.

Example

```
>>> import patito as pt
>>> relation = pt.Relation("select 1 as a union select 2 as a")
>>> relation.order(by="a").to_series()
shape: (2,)
Series: 'a' [i32]
[
            1
            2
]
```

Return type
Series

patito.Relation.union

`Relation.`**`union`**(*other*)

Produce a new relation that contains the rows of both relations.

The + operator can also be used to union two relations.

The two relations must have the same column names, but not necessarily in the same order as reordering of columns is automatically performed, unlike regular SQL.

Duplicates are *not* dropped.

Parameters

other (Union[*DataFrame*, DataFrame, DataFrame, Path, str, DuckDBPyRelation, Relation]) – A `patito.Relation` object or something that can be *casted* to `patito.Relation`. See *Relation*.

Return type

TypeVar(RelationType, bound= Relation)

Returns

New relation containing the rows of both `self` and `other`.

Raises

TypeError – If the two relations do not contain the same columns.

Examples

```
>>> import patito as pt
>>> relation_1 = pt.Relation("select 1 as a")
>>> relation_2 = pt.Relation("select 2 as a")
>>> relation_1.union(relation_2).to_df()
shape: (2, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
├╌╌╌╌╌┤
│ 2   │
└─────┘
```

```
>>> (relation_1 + relation_2).to_df()
shape: (2, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
├╌╌╌╌╌┤
│ 2   │
└─────┘
```

patito.Relation.with_columns

`Relation.`**`with_columns`**`(`*`**named_projections`*`)`

Return relations with additional columns.

If the provided columns expressions already exists as a column on the relation, the given column is overwritten.

Parameters

named_projections (`Union[str, int, float]`) – A set of column expressions, where the keyword is used as the column name, while the right-hand argument is a valid SQL expression.

Return type

`Relation`

Returns

Relation with the given columns appended, or possibly overwritten.

Examples

```
>>> import patito as pt
>>> db = pt.Database()
>>> relation = db.to_relation("select 1 as a, 2 as b")
>>> relation.with_columns(c="a + b").to_df()
shape: (1, 3)
┌─────┬─────┬─────┐
│ a   ┆ b   ┆ c   │
│ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╡
│ 1   ┆ 2   ┆ 3   │
└─────┴─────┴─────┘
```

patito.Relation.with_missing_defaultable_columns

`Relation.`**`with_missing_defaultable_columns`**`(`*`include=None`*`, `*`exclude=None`*`)`

Add missing defaultable columns filled with the default values of correct type.

Make sure to invoke *Relation.set_model()* with the correct model schema before executing `Relation.with_missing_default_columns()`.

Parameters

- **include** (`Optional[Iterable[str]]`) – If provided, only fill in default values for missing columns part of this collection of column names.
- **exclude** (`Optional[Iterable[str]]`) – If provided, do *not* fill in default values for missing columns part of this collection of column names.

Returns

New relation where missing columns with default values according to the schema have been filled in.

Return type

Relation

Example

```
>>> import patito as pt
>>> class MyModel(pt.Model):
...     non_default_column: int
...     another_non_default_column: int
...     default_column: int = 42
...     another_default_column: int = 42
...
>>> relation = pt.Relation(
...     "select 1 as non_default_column, 2 as default_column"
... )
>>> relation.to_df()
shape: (1, 2)
┌─────────────────────┬────────────────┐
│ non_default_column  ┆ default_column │
│ ---                 ┆ ---            │
│ i64                 ┆ i64            │
╞═════════════════════╪════════════════╡
│ 1                   ┆ 2              │
└─────────────────────┴────────────────┘
>>> relation.set_model(MyModel).with_missing_defaultable_columns().to_df()
shape: (1, 3)
┌─────────────────────┬────────────────┬──────────────────────┐
│ non_default_column  ┆ default_column ┆ another_default_column │
│ ---                 ┆ ---            ┆ ---                  │
│ i64                 ┆ i64            ┆ i64                  │
╞═════════════════════╪════════════════╪══════════════════════╡
│ 1                   ┆ 2              ┆ 42                   │
└─────────────────────┴────────────────┴──────────────────────┘
```

patito.Relation.with_missing_nullable_columns

`Relation.`**`with_missing_nullable_columns`**`(`*include=None*, *exclude=None*`)`

Add missing nullable columns filled with correctly typed nulls.

Make sure to invoke *Relation.set_model()* with the correct model schema before executing `Relation.with_missing_nullable_columns()`.

Parameters

- **include** (`Optional[Iterable[str]]`) – If provided, only fill in null values for missing columns part of this collection of column names.
- **exclude** (`Optional[Iterable[str]]`) – If provided, do *not* fill in null values for missing columns part of this collection of column names.

Returns

New relation where missing nullable columns have been filled in with null values.

Return type

Relation

Example

```
>>> from typing import Optional
>>> import patito as pt
>>> class MyModel(pt.Model):
...     non_nullable_column: int
...     nullable_column: Optional[int]
...     another_nullable_column: Optional[int]
...
>>> relation = pt.Relation("select 1 as nullable_column")
>>> relation.to_df()
shape: (1, 1)
┌─────────────────┐
│ nullable_column │
│ ---             │
│ i64             │
╞═════════════════╡
│ 1               │
└─────────────────┘
>>> relation.set_model(MyModel).with_missing_nullable_columns().to_df()
shape: (1, 2)
┌─────────────────┬──────────────────────┐
│ nullable_column ┆ another_nullable_col │
│ ---             ┆ ---                  │
│ i64             ┆ i64                  │
╞═════════════════╪══════════════════════╡
│ 1               ┆ null                 │
└─────────────────┴──────────────────────┘
```

patito.Relation.__add__

Relation.**__add__**(*other*)

Execute `self.union(other)`.

See *Relation.union()* for full documentation.

Return type
TypeVar(RelationType, bound= Relation)

patito.Relation.__getitem__

Relation.**__getitem__**(*key*)

Return Relation with selected columns.

Uses *Relation.select()* under-the-hood in order to perform the selection. Can technically be used to rename columns, define derived columns, and so on, but prefer the use of Relation.select() for such use cases.

Parameters
key (Union[str, Iterable[str]]) – Columns to select, either a single column represented as a string, or an iterable of strings.

Return type
Relation

Returns
New relation only containing the column subset specified.

Example

```
>>> import patito as pt
>>> relation = pt.Relation("select 1 as a, 2 as b, 3 as c")
>>> relation.to_df()
shape: (1, 3)
┌─────┬─────┬─────┐
│ a   ┆ b   ┆ c   │
│ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╡
│ 1   ┆ 2   ┆ 3   │
└─────┴─────┴─────┘
>>> relation[["a", "b"]].to_df()
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 2   │
└─────┴─────┘
>>> relation["a"].to_df()
shape: (1, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
└─────┘
```

patito.Relation.__iter__

`Relation.__iter__()`

Iterate over rows in relation.

If *Relation.set_model()* has been invoked first, the given model will be used to deserialize each row. Otherwise a Patito model is dynamically constructed which fits the schema of the relation.

Returns
An iterator of patito Model objects representing each row.

Return type
Iterator[*Model*]

Example

```
>>> from typing import Literal
>>> import patito as pt
>>> df = pt.DataFrame({"float_column": [1, 2], "enum_column": ["A", "B"]})
>>> relation = pt.Relation(df).set_alias("my_relation")
>>> for row in relation:
...     print(row)
...
float_column=1 enum_column='A'
float_column=2 enum_column='B'
>>> list(relation)
[my_relation(float_column=1, enum_column='A'),
 my_relation(float_column=2, enum_column='B')]
```

```
>>> class MySchema(pt.Model):
...     float_column: float
...     enum_column: Literal["A", "B", "C"]
...
>>> relation = relation.set_model(MySchema)
>>> for row in relation:
...     print(row)
...
float_column=1.0 enum_column='A'
float_column=2.0 enum_column='B'
>>> list(relation)
[MySchema(float_column=1.0, enum_column='A'),
 MySchema(float_column=2.0, enum_column='B')]
```

patito.Relation.__len__

Relation.**__len__**()

Return the number of rows in the relation.

See *Relation.count()* for full documentation.

Return type
int

patito.Relation.__str__

Relation.**__str__**()

Return string representation of Relation object.

Includes an expression tree, the result columns, and a result preview.

Example

```
>>> import patito as pt
>>> products = pt.Relation(
...     pt.DataFrame(
...         {
...             "product_name": ["apple", "red_apple", "banana", "oranges"],
...             "supplier_id": [2, 2, 1, 3],
...         }
...     )
... ).set_alias("products")
>>> print(str(products))  # xdoctest: +SKIP
---------------------
--- Relation Tree ---
---------------------
arrow_scan(94609350519648, 140317161740928, 140317161731168, 1000000)
---------------------
-- Result Columns  --
---------------------
- product_name (VARCHAR)
- supplier_id (BIGINT)
---------------------
-- Result Preview  --
---------------------
product_name    supplier_id
VARCHAR BIGINT
[ Rows: 4]
apple   2
red_apple       2
banana  1
oranges 3
```

```
>>> suppliers = pt.Relation(
...     pt.DataFrame(
...         {
...             "id": [1, 2],
...             "supplier_name": ["Banana Republic", "Applies Inc."],
...         }
...     )
... ).set_alias("suppliers")
>>> relation = (
...     products.set_alias("p")
...     .inner_join(
...         suppliers.set_alias("s"),
...         on="p.supplier_id = s.id",
...     )
...     .aggregate(
...         "supplier_name",
...         num_products="count(product_name)",
...         group_by=["supplier_id", "supplier_name"],
...     )
... )
>>> print(str(relation))  # xdoctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
---------------------
--- Relation Tree ---
---------------------
Aggregate [supplier_name, count(product_name)]
  Join INNER p.supplier_id = s.id
    arrow_scan(94609350519648, 140317161740928, 140317161731168, 1000000)
    arrow_scan(94609436221024, 140317161740928, 140317161731168, 1000000)
---------------------
-- Result Columns  --
---------------------
- supplier_name (VARCHAR)
- num_products (BIGINT)
---------------------
-- Result Preview  --
---------------------
supplier_name   num_products
VARCHAR BIGINT
[ Rows: 2]
Applies Inc.    2
Banana Republic 1
```

Return type
str

CHAPTER

THREE

LICENCE

Patito offers a simple way to declare pydantic data models which double as schema for your polars data frames. These schema can be used for:

Simple and performant data frame validation.

Easy generation of valid mock data frames for tests.

Retrieve and represent singular rows in an object-oriented manner.

Provide a single source of truth for the core data models in your code base.

Patito has first-class support for polars, a *"blazingly fast DataFrames library written in Rust"*.

CHAPTER

FOUR

INSTALLATION

You can simply install Patito with `pip` like so:

```
pip install patito
```

4.1 DuckDB Integration

Patito can also integrate with DuckDB. In order to enable this integration you must explicitly specify it during installation:

```
pip install 'patito[duckdb]'
```

INDEX

Symbols

A

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W