

---

**Patito**

**Jakob Gerhard Martinussen**

**Jul 21, 2023**



# CONTENTS

- 1 Tutorials 1**
  - 1.1 Using Patito for DataFrame Validation . . . . . 1
  - 1.2 Integrating Polars With an SQL Database Using Efficient Caching . . . . . 4
- 2 API Reference 15**
  - 2.1 patito.DataFrame . . . . . 15
  - 2.2 patito.Model . . . . . 23
  - 2.3 patito.Field . . . . . 39
  - 2.4 patito.Database . . . . . 40
  - 2.5 patito.duckdb . . . . . 45
- 3 Licence 89**
- 4 Installation 91**
  - 4.1 DuckDB Integration . . . . . 91
- Index 93**



## 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.

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.

`patito` is a dataframe validation library built on top of `polars` initially open sourced by Oda, which tries to solve this problem. The `polars` dataframe library has lately been making the rounds among data scientists, 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.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()`.

## 1.2 Integrating Polars With an SQL Database Using Efficient Caching

Many data-driven applications involve data that must be retrieved from a database, either a remote data warehouse solution such as Databricks or Snowflake, or a local database such as DuckDB or SQLite3. Patito offers a database-agnostic API to query such sources, returning the result as a polars `DataFrame`, while offering intelligent query caching on top. By the end of this tutorial you will be able to write data ingestion logic that looks like this:

```
from typing import Optional

from db import my_database

@my_database.as_query(cache=True)
def users(country: Optional[str] = None):
    query = "select * from users"
    if country:
        query += f" where country = '{country}'"
    return query
```

The wrapped `users` function will now construct, execute, cache, and return the results of the SQL query in the form of a `polars.DataFrame` object. The cache for `users(country="NO")` will be stored independently from `users(country="US")`, and so on. This, among with other functionality that will be explained later, allows you to integrate your local data pipeline with your remote database in an effortless way.



The following tutorial will explain how to construct a `patito.Database` object which provides Patito with the required context to execute SQL queries against your database of choice. In turn `patito.Database.query()` can be used to execute SQL query strings directly and `patito.Database.as_query()` can be used to wrap functions that *produce* SQL query strings. The latter decorator turns functions into `patito.Database.Query` objects which act very much like the original functions, only that they actually execute the constructed queries and return the results as DataFrames when invoked. The Query object also has *additional methods* for managing the query caches and more.

This tutorial will take a relatively opinionated approach to how to organize your code. For less opinionated documentation, see the referenced classes and methods above.

### Table of Contents

- *Setup*
- *Construct a patito.Database Object*
  - *Querying the Database Directly*
  - *Wrapping Query-Producing Functions*
  - *Polars vs. Pandas*
  - *Specifying custom database parameters*
  - *Normalizing column types*
- *Cache Your Queries to Speed Up Data Retrieval*
  - *Lazy data retrieval*
  - *Refreshing the cache*
  - *Automatically refreshing old caches*
  - *Specify custom cache files (advanced)*
- *Automatic Data Validation*

## 1.2.1 Setup

The following tutorial will depend on `patito` having been installed with the `caching` extension group:

```
pip install patito[caching]
```

Code samples in this tutorial will use `DuckDB`, but you should be able to replace it with your database of choice as you follow along:

```
pip install duckdb
```

## 1.2.2 Construct a `patito.Database` Object

To begin we need to provide Patito with the tools required to query your database of choice. First we must implement a *query handler*, a function that takes a query string as its first argument, executes the query, and returns the result of the query in the form an Arrow table.

We are going to use DuckDB as our detailed example in this tutorial, but example code for other databases, including SQLite3, is provided at the end of this section. We start by creating a `db.py` module in the root of our application, and implement `db.connection` as a way to connect to a DuckDB instance.

Listing 4: `db.py` – connection

```
import duckdb

def connection(name: str) -> duckdb.DuckDBPyConnection:
    return duckdb.connect(name)
```

Here `db.connection()` must be provided with a name, either `:memory:` to store the data in-memory, or a file name to persist the data on-disk. We can use this new function in order to implement our query handler.

Listing 5: `db.py` - query\_handler

```
import duckdb
import pyarrow as pa

def connection(name: str) -> duckdb.DuckDBPyConnection:
    return duckdb.connect(name)

def query_handler(query: str, *, name: str = ":memory:") -> pa.Table:
    connection = connection(name=name)
    return connection.cursor().query(query).arrow()
```

Notice how the first argument of `query_handler` is the query string to be executed, as required by Patito, but the `name` keyword is specific to our database of choice. It is now simple for us to create a `patito.Database` object by providing `db.query_handler`:

Listing 6: `db.py` – `pt.Database`

```
from pathlib import Path

import duckdb
import patito as pt
import pyarrow as pa

def connection(name: str) -> duckdb.DuckDBPyConnection:
    return duckdb.connect(name)

def query_handler(query: str, name: str = ":memory:") -> pa.Table:
    cursor = connection(name).cursor()
    return cursor.query(query).arrow()

my_database = pt.Database(query_handler=query_handler)
```

Additional arguments can be provided to the Database constructor, for example a custom cache directory. These additional parameters are documented [here](#). Documentation for constructing query handlers and `patito.Database` objects for other databases is provided in the collapsable sections below:

See “Examples” section of `patito.Database`.

You are welcome to create a [GitHub issue](#) if you need help integrating with your specific database of choice.

## Querying the Database Directly

The `db` module is now complete and we should be able to use it in order to execute queries directly against our in-memory database.

```
>>> from db import my_database
>>> my_database.query("select 1 as a, 2 as b")
shape: (1, 2)
```

a	b
---	---
i32	i32
1	2

The query result is provided in the form of a polars DataFrame object. Additional parameters can be provided to `patito.Database.query()` as described [here](#). As an example, the query result can be provided as a polars LazyFrame by specifying `lazy=True`.

```
>>> from db import my_database
>>> my_database.query("select 1 as a, 2 as b", lazy=True)
<polars.LazyFrame object at 0x13571D310>
```

Any *additional* keyword arguments provided to `patito.Database.query()` are forwarded directly to the original query handler, so the following will execute the query against the database stored in `my.db`:

```
>>> my_database.query("select * from my_table", name="my.db")
```

Fig. 1: Delegation of parameters provided to `patito.Database.query()`.

## Wrapping Query-Producing Functions

Let’s assume that you have a project named `user-analyzer` which analyzes users. The associated python package should therefore be named `user_analyzer`. By convention, functions for retrieving data from a remote database should be placed in the `user_analyzer.fetch` sub-module. Using this module should be as simple as...

```
from user_analyzer import fetch

user_df = fetch.users()
```

Start by creating the python file for the `fetch` sub-module, it should be located at `projects/user-analyzer/user_analyzer/fetch.py`. Next, implement the `users` function as a function that returns a SQL query that should produce the intended data when executed in the remote database...

```
def users():  
    return "select * from d_users"
```

This is clearly not enough, the `fetch.users` function only returns a query string for now, but it can trivially be converted to a function that returns a dataframe instead by using the query decorator from `db`...

```
from db import query  
  
@query()  
def users():  
    return "select * from d_users"
```

## Polars vs. Pandas

When `user_analyzer.fetch.users()` is invoked it will return a polars `DataFrame` by default. [Polars](#) is a `DataFrame` library that is highly recommended over `pandas`; it will be familiar to most `pandas` users and can be easily converted to `pandas` when needed. You can find introductory documentation for `polars` [here](#). If you still prefer to use `pandas` you can use the `.to_pandas()` method like this...

```
from user_analyzer import fetch  
  
# This returns a polars DataFrame  
user_df = fetch.users()  
  
# This returns a pandas DataFrame  
user_df = fetch.users().to_pandas()
```

We can also add parameters to the `users` function, if needed, let's say we want to be able to filter on the users' country codes:

```
from typing import Literal, Optional  
  
from db import query  
  
@query()  
def users(country: Optional[str] = None):  
    if country_code:  
        return f"select * from d_users where country_code = '{country}'"  
    else:  
        return "select * from d_users"
```

You can now construct a `DataFrame` of all Finish users by writing `fetch.users(country="FI")`. If you want to access the SQL query rather than executing it, you can retrieve it with `fetch.users.query_string(country="FI")`.

## Specifying custom database parameters

The `@query` decorator will by default execute your SQL query against the `ANALYTICS.ANALYTICS` database schema. If your query needs to use different schema, warehouses, users, etc., you can specify a custom `db_params` parameter to the query decorator.

Here is an example where we execute the query against `ANALYTICS.ANALYTICS_FORECASTING` instead of `ANALYTICS.ANALYTICS`.

```
from db import query

FORECASTING_SCHEMA = {"schema": "ANALYTICS_FORECASTING"}

@query(db_params=FORECASTING_SCHEMA):
def covid_cases():
    return "return * from stg_covid_cases"
```

## Normalizing column types

A Snowflake query might produce different column types based on how many rows are returned and/or the value bounds of each column. In order to ensure consistent behavior, `db.query` by default `_upcasts_` all lower-typed dtypes such as `Int8` to `Int64`, `Float16` to `Float64`, and so on. This behavior can be disabled by providing `normalize_column_types=False` to the `@query` decorator.

```
from db import query

@query(normalize_column_types=False)
def example_query():
    return "example query"
```

## 1.2.3 Cache Your Queries to Speed Up Data Retrieval

Some database queries may take a long time to execute due to the data set being large and/or the computations being intensive. In those cases you might want to store the result for reuse rather than re-executing the query every single time you invoke `fetch.X()`. Luckily, this is really easy with `db.query`, you can simply add the `query=True` parameter to the decorator and caching will be automatically enabled!

Enabling caching for `fetch.users` will look like this...

```
...

@query(cache=True)
def users(country: Optional[str] = None):
    ...
```

Now, if you execute `fetch.users()` it will query the database directly, but the `_next_` time you execute it, it will instantaneously return the result from the previous execution. The `@query` decorator will cache the results based on the query string itself, so `fetch.users()`, `fetch.users(country="FI")`, `fetch.users(country="NO")`, and so on will be cached independently.

## Lazy data retrieval

You can also specify the `lazy=True` parameter to the `@query` decorator in order to receive the query result in the form of a `LazyFrame` object rather than a `DataFrame`. This parameter plays well with cached query decorators since it will only read the *strictly required* data from the cache.

```
...

@query(cache=True, lazy=True)
def users():
    ...

# Only the subset of the rows with age_in_years >= 67 will be read into memory
pensioners = users().filter(pl.col("age_in_years") >= 67).collect()
```

## Refreshing the cache

Sometimes you may want to forcefully reset the cache of a query function in order to get the latest version of the data from remote database. This can be done by invoking `X.refresh_cache()` rather than `X()` directly. Let's say you want to retrieve the latest set of Norwegian users from the database...

```
from user_analyzer import fetch

user_df = fetch.users.refresh_cache(country="NO")
```

This will delete the cached version of the Norwegian users if the result has already been cached, and return the latest result. The next time you invoke `fetch.users(country="NO")` you will get the latest version of the cache. If you want to clear *all* caches, regardless query parameterization, you can use the `X.clear_caches()` method.

```
from user_analyzer import fetch

fetch.users.clear_caches()
```

The `.refresh_cache()` and `.clear_caches()` methods are in fact part of several other methods that are automatically added to `@query`-decorated functions, the full list of such methods is:

- `.clear_caches()` - Delete all cache files of the given query function such that new data will be fetched the `_next_` time the query is invoked.
- `.refresh_cache(*args, **kwargs)` - Force the resulting SQL query produced by the given parameters to be executed in the remote database and repopulate the parameter-specific cache.
- `.cache_path(*args, **kwargs)` - Return a `pathlib.Path` object pointing to the parquet file that is used to store the cache for the given parameters.
- `.query_string(*args, **kwargs)` - Return the SQL query string to be executed.

## Automatically refreshing old caches

Sometimes it makes sense to cache a query result, but not *forever*. In such cases you can specify the *Time to Live* (TTL) of the cache, automatically refreshing the cache when it becomes older than the specified TTL. This can be done by specifying the `ttl` argument to the `@query` decorator as a `datetime.timedelta`.

Let's say that we want to fetch the newest collection of users once a day, but otherwise cache the results. This can be achieved in the following way...

```
from datetime import timedelta

from db import query

@query(
    cache=True,
    ttl=timedelta(days=1),
)
def users(country: Optional[str] = None):
    ...
```

The first time you invoke `fetch.users()`, the query will be executed in the remote database and the result will be cached. After that, the cache will be used until you invoke `fetch.users()` more than 24 hours after the cache was initially created. Then the cache will be automatically refreshed. You can also force a cache refresh any time by using the `.refresh_cache()` method, for instance for all Norwegian users by executing `fetch.users.refresh_cache(country="NO")`.

## Specify custom cache files (advanced)

If you want to store the cached results in specific parquet files, you can specify the `cache` parameter to the `@query` decorator as a string or as a `pathlib.Path` object. Let's say you want to store the users in a file called `users.parquet`, this can be done in the following way:

```
from db import query

@query(cache="users.parquet")
def users(country: Optional[str] = None):
    ...
```

The file path `users.parquet` is a so-called *relative path* and is therefore interpreted relative the `artifacts/query_cache` sub-directory within the project's root. You can inspect the resulting path by executing `users.cache_path()`:

```
from user_analyzer import fetch

print(fetch.users.cache_path())
# Outputs: /repo/projects/user-analyzer/artifacts/query_cache/users.parquet
```

You can also specify an absolute path if required, let's say you want to place the file in `<REPO>/projects/user-analyzer/users.parquet`:

```
from db import PROJECT_DIR, query

@query(cache=PROJECT_DIR / "users.parquet")
```

(continues on next page)

(continued from previous page)

```
def users(country: Optional[str] = None):
    ...
```

The problem with the previous custom cache path is that `fetch.users(country="NO")` and `fetch.users(countnr="FI")` will write to the same cache file, thus refreshing the cache much more than strictly necessary. It would be more efficient to have a separate cache file for each country. You can achieve this by inserting a `{country}` formatting placeholder, like with an f-string, in the custom cache path:

```
from db import PROJECT_DIR, query

@query(cache=PROJECT_DIR / "users-{country}.parquet")
def users(country: Optional[str] = None):
    ...
```

Finish users will now be cached in `users-FI.parquet`, while Norwegian users will be cached in `users-NO.parquet`.

## 1.2.4 Automatic Data Validation

The `@query` decorator integrates with the `patito` DataFrame validation library, allowing you to automatically validate the data fetched from the remote database. If the concept of data validation, and why you should apply it in your data science projects, is new to you, then you should read “[Using Patito for DataFrame Validation](#)”.

Let’s say that we have a `fetch.products()` query function which produces a DataFrame of three columns.

```
from db import query

@query()
def products():
    return """
        select
            product_id,
            warehouse_department,
            current_retail_price

        from products
    """
```

Given this query we might want to validate the following assumptions:

- `product_id` is a unique integer assigned to each product.
- `warehouse_department` takes one of three permissible values: "Dry", "Cold", or "Frozen".
- `current_retail_price` is a positive floating point number.

By convention we should define a Patito model class named `Product` placed in `<project_module>/models.py`.

```
import patito as pt

class Product(pt.Model):
    product_id: int = pt.Field(unique=True)
    warehouse_department: Literal["Dry", "Cold", "Frozen"]
    current_retail_price: float = pt.Field(gt=0)
```



We can now use `user_analyzer.models.Product` to automatically validate the data produced by `user_analyzer.fetch.products` by providing the `model` keyword to the `@query` decorator.

```
from db import query

from user_analyzer import models

@query(model=models.Product)
def products():
    return """
        select
            product_id,
            warehouse_department,
            current_retail_price

        from products
    """
```

Whenever you invoke `fetch.products`, the data will be guaranteed to follow the schema of `models.Product`, otherwise an exception will be raised. You can therefore rest assured that the production data will not substantially change without you noticing it in the future.



## API REFERENCE

### 2.1 patito.DataFrame

```
class patito.DataFrame(data=None, schema=None, *, schema_overrides=None, orient=None,  
                      infer_schema_length=100, nan_to_null=False)
```

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 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

## 2.1.2 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
1	1	2
2	2	4

### 2.1.3 patito.DataFrame.drop

`DataFrame.drop(columns=None, *more_columns)`

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, Collection[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.
- **more\_columns** (str) – Additional named columns to drop.

#### 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
1
2

(continues on next page)

(continued from previous page)

1
2

## 2.1.4 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

## 2.1.5 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.
```

## 2.1.6 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` parameter 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 | # |
```

## 2.1.7 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)
...
>>> classes = pt.DataFrame(
...     {"year": [1, 1, 2, 2], "letter": list("ABAB")}
... ).set_model(SchoolClass)
>>> classes
shape: (4, 2)
```

(continues on next page)

(continued from previous page)

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()
```

## 2.1.8 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 patito.Model.DataFrame

`Model.DataFrame(data=None, schema=None, *, schema_overrides=None, orient=None, infer_schema_length=100, nan_to_null=False)`

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 `Product.validate`.

### 2.2.2 patito.Model.LazyFrame

`Model.LazyFrame`(*data=None, schema=None, \*, schema\_overrides=None, orient=None, infer\_schema\_length=100, nan\_to\_null=False*)

`LazyFrame` class associated to `DataFrame`.

### 2.2.3 patito.Model.columns

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

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

**Returns**

List of column names.

**Example**

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

### 2.2.4 patito.Model.defaults

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

Return default field values specified on the model.

**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'}

```

## 2.2.5 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']

```

## 2.2.6 patito.Model.dtypes

**property** `Model.dtypes`: `dict[str, Type[polars.datatypes.classes.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.

**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': Utf8, 'ideal_temperature': Int64, 'price': Float64}
```

## 2.2.7 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')
```

## 2.2.8 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
>>> Product.example_value("name")
'dummy_string'
>>> Product.example_value("temperature_zone")
'dry'
```

## 2.2.9 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
---	---	---

(continues on next page)



(continued from previous page)

str	cat	i64
product A	dry	0
product B	dry	1

## 2.2.10 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"}],
...     schema=["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')
```

## 2.2.11 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
```

### 2.2.12 patito.Model.non\_nullable\_columns

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

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

**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']
```

### 2.2.13 patito.Model.nullable\_columns

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

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

**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']
```

## 2.2.14 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
```

## 2.2.15 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']
```

**2.2.16 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']
```

**2.2.17 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']
```

## 2.2.18 patito.Model.sql\_types

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

Return compatible DuckDB SQL types for all model fields.

**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'}
```

### 2.2.19 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']
```

### 2.2.20 patito.Model.unique\_columns

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

Return columns with uniqueness constraint.

**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']
```

### 2.2.21 patito.Model.valid\_dtypes

**property** `Model.valid_dtypes`: `dict[str, List[pl.PolarsDataType | pl.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.

**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': [Boolean],
 'float_column': [Float64, Float32],
 'int_column': [Int64, Int32, Int16, Int8, UInt64, UInt32, UInt16, UInt8],
 'str_column': [Utf8]}
```

### 2.2.22 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', 'TIMESTAMP TZ', '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.

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

```

### 2.2.23 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

### Return type

None

```
>>> 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)

## 2.2.24 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** (`Optional[float]`) – All values must be greater than `gt`.
- **ge** (`Optional[float]`) – All values must be greater than or equal to `ge`.
- **lt** (`Optional[float]`) – All values must be less than `lt`.
- **le** (`Optional[float]`) – All values must be less than or equal to `lt`.
- **multiple\_of** (`Optional[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 Int64 does not match model field type. (type=type_error.
  ↳columndtype)
brand_color
  2 rows with out of bound values. (type=value_error.rowvalue)
```

## 2.4 patito.Database

**class** patito.Database(query\_handler, cache\_directory=None, default\_ttl=datetime.timedelta(days=364))

Construct manager for executing SQL queries and caching the results.

### Parameters

- **query\_handler** (Callable[... Table]) – The function that the Database object should use for executing SQL queries. Its first argument should be the SQL query string to execute, and it should return the query result as an arrow table, for instance pyarrow.Table.

- **cache\_directory** (Optional[Path]) – Path to the directory where caches should be stored as parquet files. If not provided, the [XDG Base Directory Specification](#) will be used to determine the suitable cache directory, by default `~/.cache/patito` or `${XDG_CACHE_HOME}/patito`.
- **default\_ttl** (timedelta) – The default Time To Live (TTL), or with other words, how long to wait until caches are refreshed due to old age. The given default TTL can be overwritten by specifying the `ttl` parameter in `Database.query()`. The default is 52 weeks.

## Examples

We start by importing the necessary modules:

```
>>> from pathlib import Path
...
>>> import patito as pt
>>> import pyarrow as pa
```

In order to construct a Database, we need to provide the constructor with a function that can *execute* query strings. How to construct this function will depend on what you actually want to run your queries against, for example a local or remote database. For the purposes of demonstration we will use SQLite since it is built into Python's standard library, but you can use anything; for example Snowflake or PostgreSQL.

We will use Python's standard library [documentation](#) to create an in-memory SQLite database. It will contain a single table named `movies` containing some dummy data. The details do not really matter here, the only important part is that we construct a database which we can run SQL queries against.

```
>>> import sqlite3
...
>>> def dummy_database() -> sqlite3.Cursor:
...     connection = sqlite3.connect(":memory:")
...     cursor = connection.cursor()
...     cursor.execute("CREATE TABLE movies(title, year, score)")
...     data = [
...         ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
...         ("Monty Python's The Meaning of Life", 1983, 7.5),
...         ("Monty Python's Life of Brian", 1979, 8.0),
...     ]
...     cursor.executemany("INSERT INTO movies VALUES(?, ?, ?)", data)
...     connection.commit()
...     return cursor
```

Using this dummy database, we are now able to construct a function which accepts SQL queries as its first parameter, executes the query, and returns the query result in the form of an Arrow table.

```
>>> def query_handler(query: str) -> pa.Table:
...     cursor = dummy_database()
...     cursor.execute(query)
...     columns = [description[0] for description in cursor.description]
...     data = [dict(zip(columns, row)) for row in cursor.fetchall()]
...     return pa.Table.from_pylist(data)
```

We can now construct a Database object, providing `query_handler` as the way to execute SQL queries.

```
>>> db = pt.Database(query_handler=query_handler)
```

The resulting object can now be used to execute SQL queries against the database and return the result in the form of a polars DataFrame object.

```
>>> db.query("select * from movies order by year limit 1")
shape: (1, 3)
```

title	year	score
---	---	---
str	i64	f64
Monty Python's Life of Brian	1979	8.0

But the main way to use a Database object is to use the `@Database.as_query` decarator to wrap functions which return SQL query *strings*.

```
>>> @db.as_query()
>>> def movies(newer_than_year: int):
...     return f"select * from movies where year > {newer_than_year}"
```

This decorator will convert the function from producing query strings, to actually executing the given query and return the query result in the form of a polars DataFrame object.

```
>>> movies(newer_than_year=1980)
shape: (2, 3)
```

title	year	score
---	---	---
str	i64	f64
Monty Python Live at the Hollywo...	1982	7.9
Monty Python's The Meaning of Li...	1983	7.5

Caching is not enabled by default, but it can be enabled by specifying `cache=True` to the `@db.as_query(...)` decorator. Other arguments are also accepted, such as `lazy=True` if you want to retrieve the results in the form of a `LazyFrame` instead of a `DataFrame`, `t1l` if you want to specify another TTL, and any additional keyword arguments are forwarded to `query_executor` when the SQL query is executed. You can read more about these parameters in the documentation of [Database.query\(\)](#).

## 2.4.1 patito.Database.Query

```
class patito.database.DatabaseQuery(query_constructor, cache_directory, query_handler, ttl, lazy=False,
                                    cache=False, model=None, query_handler_kwargs=None)
```

A class acting as a function that returns a `polars.DataFrame` when called.

**patito.Database.Query.cache\_path**

Database.Query.cache\_path(\*args, \*\*kwargs)

Return the path to the parquet cache that would store the result of the query.

**Parameters**

- **args** (ParamSpecArgs) – The positional arguments passed to the wrapped function.
- **kwargs** (ParamSpecKwargs) – The keyword arguments passed to the wrapped function.

**Return type**

Optional[Path]

**Returns**

A deterministic path to a parquet cache. None if caching is disabled.

**patito.Database.Query.clear\_caches**

Database.Query.clear\_caches()

Delete all parquet cache files produced by this query wrapper.

**Return type**

None

**patito.Database.Query.query\_string**

Database.Query.query\_string(\*args, \*\*kwargs)

Return the query to be executed for the given parameters.

**Parameters**

- **\*args** (ParamSpecArgs) – Positional arguments used to construct the query string.
- **\*kwargs** (ParamSpecKwargs) – Keyword arguments used to construct the query string.

**Return type**

str

**Returns**

The query string produced for the given input parameters.

**patito.Database.Query.refresh\_cache**

Database.Query.refresh\_cache(\*args, \*\*kwargs)

Force query execution by refreshing the cache.

**Parameters**

- **\*args** (ParamSpecArgs) – Positional arguments used to construct the SQL query string.
- **\*kwargs** (ParamSpecKwargs) – Keyword arguments used to construct the SQL query string.

**Return type**

TypeVar(DF, bound= Union[DataFrame, LazyFrame], covariant=True)

**Returns**

A DataFrame representing the result of the newly executed query.

## patito.Database.Query.\_\_call\_\_

Database.Query.\_\_call\_\_(\*args, \*\*kwargs)

Call self as a function.

### Return type

TypeVar(DF, bound= Union[DataFrame, LazyFrame], covariant=True)

## 2.4.2 patito.Database.as\_query

Database.as\_query(\*, lazy=False, cache=False, ttl=None, model=None, \*\*kwargs)

Execute the returned query string and return a polars dataframe.

### Parameters

- **lazy** (bool) – If the result should be returned as a LazyFrame rather than a DataFrame. Allows more efficient reading from parquet caches if caching is enabled.
- **cache** (Union[str, Path, bool]) – If queries should be cached in order to save time and costs. The cache will only be used if the exact same SQL string has been executed before. If the parameter is specified as True, a parquet file is created for each unique query string, and is located at: artifacts/query\_cache/<function\_name>/<query\_md5\_hash>.parquet If the a string or pathlib.Path object is provided, the given path will be used, but it must have a '.parquet' file extension. Relative paths are interpreted relative to artifacts/query\_cache/ in the workspace root. The given parquet path will be overwritten if the query string changes, so only the latest query string value will be cached.
- **ttl** (Optional[timedelta]) – The Time to Live (TTL) of the cache specified as a date-time.timedelta object. When the cache becomes older than the specified TTL, the query will be re-executed on the next invocation of the query function and the cache will refreshed.
- **model** (Optional[Type[Model]]) – An optional Patito model used to validate the content of the dataframe before return.
- **\*\*kwargs** (Any) – Connection parameters forwarded to sql\_to\_polars, for example db\_params.

### Return type

Callable[[QueryConstructor[ParamSpec(P)]], [DatabaseQuery](#)[ParamSpec(P), Union[DataFrame, LazyFrame]]]

### Returns

A new function which returns a polars DataFrame based on the query specified by the original function's return string.

## 2.4.3 patito.Database.query

Database.query(query, \*, lazy=False, cache=False, ttl=None, model=None, \*\*kwargs)

Execute the given query and return the query result as a DataFrame or LazyFrame.

See [patito.Database.as\\_query](#) for a more powerful way to build and execute queries.

### Parameters

- **query** (str) – The query string to be executed, for instance an SQL query.
- **lazy** (bool) – If the query result should be returned in the form of a LazyFrame instead of a DataFrame.



- **cache** (Union[str, Path, bool]) – If the query result should be saved and re-used the next time the same query is executed. Can also be provided as a path. See [Database.as\\_query\(\)](#) for full documentation.
- **ttl** (Optional[timedelta]) – How long to use cached results until the query is re-executed anyway.
- **model** (Optional[Type[Model]]) – A *patito.Model* to optionally validate the query result.
- **\*\*kwargs** (Any) – All additional keyword arguments are forwarded to the query handler which executes the given query.

**Return type**

Union[DataFrame, LazyFrame]

**Returns**

The result of the query in the form of a DataFrame if lazy=False, or a LazyFrame otherwise.

**Examples**

We will use DuckDB as our example database.

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

We will construct a really simple query source from an in-memory database.

```
>>> db = duckdb.connect(":memory:")
>>> query_handler = lambda query: db.cursor().query(query).arrow()
>>> query_source = pt.Database(query_handler=query_handler)
```

We can now use [Database.query\(\)](#) in order to execute queries against the in-memory database.

```
>>> query_source.query("select 1 as a, 2 as b, 3 as c")
shape: (1, 3)
```

a	b	c
i32	i32	i32
1	2	3

## 2.5 patito.duckdb

### 2.5.1 patito.duckdb.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.duckdb.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

## patito.duckdb.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.

### Return type

None

## Example

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

**patito.duckdb.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.duckdb.Database()
>>> db.create_table(name="my_table", model=MyModel)
>>> db.table("my_table").types
{'str_column': VARCHAR, 'nullable_string_column': VARCHAR}
```

**patito.duckdb.Database.create\_view**

`Database.create_view(name, data)`

Create a view based on the given data source.

**Return type**

Relation

**patito.duckdb.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.duckdb.Database.default()
>>> db.query("select 1 as a, 2 as b").to_df()
shape: (1, 2)
```

a	b
1	2

**patito.duckdb.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.

**Return type**

Relation[TypeVar(ModelType, bound= Model)]

## Example

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

(continues on next page)

(continued from previous page)

str	bool
dummy	true

**patito.duckdb.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.

**Return type**

None

**Example**

```
>>> import patito as pt
>>> db = pt.duckdb.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
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

(continues on next page)

(continued from previous page)

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

### patito.duckdb.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.duckdb.Database.from_connection(connection)
```

### patito.duckdb.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.

**Return type**  
Relation

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

### Example

```
>>> import patito as pt
>>> db = pt.duckdb.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

### patito.duckdb.Database.table

Database.**table**(*name*)

Return relation representing all the data in the given table.

**Parameters**  
**name** (str) – The name of the table.

**Return type**  
Relation

### Example

```
>>> import patito as pt
>>> df = pt.DataFrame({"a": [1, 2], "b": [3, 4]})
>>> db = pt.duckdb.Database()
>>> relation = db.to_relation(df)
>>> relation.create_table(name="my_table")
>>> db.table("my_table").to_df()
shape: (2, 2)
```

(continues on next page)

(continued from previous page)

a	b
---	---
i64	i64
1	3
2	4

## patito.duckdb.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.

### Return type

Relation

## Example

```
>>> import patito as pt
>>> db = pt.duckdb.Database()
>>> db.to_relation("select 1 as a, 2 as b").to_df()
shape: (1, 2)
```

a	b
---	---
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



**patito.duckdb.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.**Return type**

bool

**Examples**

```
>>> import patito as pt
>>> db = pt.duckdb.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
```

**2.5.2 patito.duckdb.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.duckdb.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.duckdb.Relation(df).filter("a > 2").to_df()
shape: (1, 2)
```

a	b
i64	i64
3	6

Instantiated from an SQL query:

```
>>> pt.duckdb.Relation("select 1 as a, 2 as b").to_df()
shape: (1, 2)
```

a	b
i64	i64
1	2

## patito.duckdb.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 re-named.
- **exclude** (Optional[Iterable[str]]) – If provided, the given columns will *not* be re-named.

### Raises

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

### Return type

Relation

## Examples

```
>>> import patito as pt
>>> relation = pt.duckdb.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)
```

column_1	renamed_column_2
i64	i64
1	2

### patito.duckdb.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 re-named.
- **exclude** (Optional[Collection[str]]) – If provided, the given columns will *not* be re-named.

#### Raises

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

#### Return type

Relation

## Examples

```
>>> import patito as pt
>>> relation = pt.duckdb.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

## patito.duckdb.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.

### Return type

Relation

## Examples

```
>>> import patito as pt
>>> df = pt.DataFrame({"a": [1, 2, 3], "b": ["X", "Y", "X"]})
>>> relation = pt.duckdb.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

### patito.duckdb.Relation.alias

The alias that can be used to refer to the given relation in queries. Can be set with [Relation.set\\_alias\(\)](#).

### patito.duckdb.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.

#### Return type

bool

## Examples

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

(continues on next page)

(continued from previous page)

```

>>> relation = pt.duckdb.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

```

### patito.duckdb.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.

#### Return type

Relation

### Examples

The following case statement...

```

>>> import patito as pt
>>> db = pt.duckdb.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
---	---

(continues on next page)

(continued from previous page)

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

## patito.duckdb.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.duckdb.Relation("select 1 as float_column")
>>> relation.types["float_column"]
INTEGER
>>> relation.cast(model=Schema).types["float_column"]
DOUBLE
```

```
>>> relation = pt.duckdb.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.duckdb.Relation(
...     "select 1 as column_1, 2 as column_2"
... ).set_model(Schema)
>>> 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.duckdb.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.duckdb.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.duckdb.Relation.columns

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

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

## Examples

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

### patito.duckdb.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.duckdb.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.duckdb.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.duckdb.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.duckdb.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.duckdb.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
A
A
B

**patito.duckdb.Relation.distinct****Relation.distinct**()

Drop all duplicate rows of the relation.

**Return type**

TypeVar(RelationType, bound= Relation)

**Example**

```
>>> import patito as pt
>>> df = pt.DataFrame(
...     [[1, 2, 3], [1, 2, 3], [3, 2, 1]],
...     schema=["a", "b", "c"],
...     orient="row",
... )
>>> relation = pt.duckdb.Relation(df)
>>> relation.to_df()
shape: (3, 3)
```

a	b	c
1	2	3
1	2	3
3	2	1

(continues on next page)

(continued from previous page)

---	---	---
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

## patito.duckdb.Relation.drop

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

Remove specified column(s) from relation.

### Parameters

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

### Return type

Relation

## Examples

```
>>> import patito as pt
>>> relation = pt.duckdb.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']
```

## patito.duckdb.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.duckdb.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.duckdb.Relation("select 2")
>>> relation_2.to_df()
shape: (1, 1)
┌───┐
│ 2 │
│ --- │
│ i64 │
└───┘

┌───┐
│ 2 │
└───┘

>>> relation_123.except_(relation_2).order(by="1").to_df()
shape: (2, 1)
┌───┐
│ 1 │
│ --- │
│ i64 │
└───┘

┌───┐
│ 1 │
│ 3 │
└───┘
```

**patito.duckdb.Relation.execute****Relation.execute()**

Execute built relation query and return result object.

**Return type**

DuckDBPyRelation

**Returns**

A native duckdb.DuckDBPyResult object representing the executed query.

## Examples

```
>>> import patito as pt
>>> relation = pt.duckdb.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.duckdb.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.duckdb.Relation(df)
>>> relation.filter("number % 2 = 0").to_df()
shape: (2, 2)
```

number	string
2	A
4	B

```
>>> relation.filter(number=1, string="A").to_df()
shape: (1, 2)
```

number	string
---	---
i64	str
1	A

## patito.duckdb.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.duckdb.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.duckdb.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.duckdb.Relation(products_df)
>>> supplier_df = pt.DataFrame(
...     {
```

(continues on next page)



(continued from previous page)

```

...     "id": [1, 2],
...     "supplier_name": ["Banana Republic", "Applies Inc."],
...     }
... )
>>> suppliers = pt.duckdb.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.duckdb.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.duckdb.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()

```

(continues on next page)

(continued from previous page)

shape: (2, 1)

a
i64
1
2

**patito.duckdb.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.duckdb.Relation(df1).intersect(pt.duckdb.Relation(df2)).to_df()
shape: (1, 2)
```

a	b
i64	i64
1	1

**patito.duckdb.Relation.join**

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

Join relation with other relation source based on condition.

See [duckdb.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.duckdb.Relation(products_df)
>>> supplier_df = pt.DataFrame(
...     {
...         "id": [1, 2],
...         "supplier_name": ["Banana Republic", "Applies Inc."],
...     }
... )
>>> suppliers = pt.duckdb.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.duckdb.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.duckdb.Relation(products_df)
>>> supplier_df = pt.DataFrame(
...     {
...         "id": [1, 2],
...         "supplier_name": ["Banana Republic", "Applies Inc."],
...     }
... )
>>> suppliers = pt.duckdb.Relation(supplier_df)
>>> products.set_alias("p").left_join(
```

(continues on next page)

(continued from previous page)

```
...     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.duckdb.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.duckdb.Relation("select 1 as column")
...     + pt.duckdb.Relation("select 2 as column")
...     + pt.duckdb.Relation("select 3 as column")
...     + pt.duckdb.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)
```

(continues on next page)

(continued from previous page)

column
---
i64
3
4

**patito.duckdb.Relation.model**

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

**patito.duckdb.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.duckdb.Relation(df)
```

(continues on next page)

(continued from previous page)

```
>>> relation.order(by="age desc").to_df()
shape: (4, 2)
```

name	age
---	---
str	i64
Diana	35
Charles	30
Alice	20
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.duckdb.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

### Return type

Relation

```
>>> import patito as pt
>>> relation = pt.duckdb.Relation("select 1 as a, 2 as b")
>>> relation.rename(b="c").to_df().select(["a", "c"])
shape: (1, 2)
```

a	c
---	---
i64	i64

(continues on next page)

1	2
---	---

## patito.duckdb.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 or 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".

### Return type

Relation

## Examples

```
>>> import patito as pt
>>> db = pt.duckdb.Database()
>>> relation = db.to_relation(pt.DataFrame({"original_column": [1, 2, 3]}))
>>> relation.select("*").to_df()
shape: (3, 1)
```

original_column
---
i64
1
2
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



**patito.duckdb.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.duckdb.Relation("select 1 as a, 2 as b")
>>> relation_2 = pt.duckdb.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.duckdb.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.duckdb.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.duckdb.Relation.to\_df

#### Relation.to\_df()

Return a polars DataFrame representation of relation object.

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

#### Return type

*DataFrame*

### Example

```

>>> import patito as pt
>>> pt.duckdb.Relation("select 1 as column union select 2 as column").order(
...     by="1"
... ).to_df()
shape: (2, 1)

```

column
i64
1
2

**patito.duckdb.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.

**Return type**

DataFrame

**Example**

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

**patito.duckdb.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.

**Return type**

Series

**Example**

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

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

**patito.duckdb.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.duckdb.Relation` object or something that can be *casted* to `patito.duckdb.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.duckdb.Relation("select 1 as a")
>>> relation_2 = pt.duckdb.Relation("select 2 as a")
>>> relation_1.union(relation_2).to_df()
shape: (2, 1)
```

a
---
i64

(continues on next page)

(continued from previous page)

1
2

```
>>> (relation_1 + relation_2).to_df()
shape: (2, 1)
```

a
---
i64
1
2

### patito.duckdb.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 key-word 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.duckdb.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.duckdb.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_defaultable_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.duckdb.Relation(
...     "select 1 as non_default_column, 2 as default_column"
... )
>>> relation.to_df()
shape: (1, 2)
```

non_default_column	default_column
1	2

```
>>> relation.set_model(MyModel).with_missing_defaultable_columns().to_df()
shape: (1, 3)
```

non_default_column	default_column	another_default_column
1	2	42

**patito.duckdb.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.duckdb.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_column
i64	i64
1	null

## patito.duckdb.Relation.\_\_add\_\_

Relation.\_\_add\_\_(other)

Execute self.union(other).

See [Relation.union\(\)](#) for full documentation.

### Return type

TypeVar(RelationType, bound= Relation)

## patito.duckdb.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.duckdb.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  │
└───┘
```

(continues on next page)



(continued from previous page)

---
i64
1

**patito.duckdb.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.duckdb.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.duckdb.Relation.\_\_len\_\_****Relation.\_\_len\_\_()**

Return the number of rows in the relation.

See *Relation.count()* for full documentation.**Return type**  
int**patito.duckdb.Relation.\_\_str\_\_****Relation.\_\_str\_\_()**

Return string representation of Relation object.

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

**Return type**  
str**Example**

```
>>> import patito as pt
>>> products = pt.duckdb.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, 10000000)
-----
-- 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.duckdb.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
-----
--- Relation Tree ---
-----
Aggregate [supplier_name, count(product_name)]
  Join INNER p.supplier_id = s.id
    arrow_scan(94609350519648, 140317161740928, 140317161731168, 10000000)
    arrow_scan(94609436221024, 140317161740928, 140317161731168, 10000000)
-----
-- Result Columns --
-----
- supplier_name (VARCHAR)
- num_products (BIGINT)
-----
-- Result Preview --
-----
supplier_name  num_products
VARCHAR BIGINT
[ Rows: 2]
Applies Inc.   2
Banana Republic 1

```



## LICENCE

### MIT License

Copyright (c) 2022 Oda Group Holding AS Copyright (c) 2023 Jakob Gerhard Martinussen and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

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*”.



## 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]'
```





## Symbols

\_\_add\_\_() (*patito.duckdb.Relation method*), 84  
 \_\_call\_\_() (*patito.Database.Query method*), 44  
 \_\_contains\_\_() (*patito.duckdb.Database method*), 53  
 \_\_getitem\_\_() (*patito.duckdb.Relation method*), 84  
 \_\_init\_\_() (*patito.duckdb.Database method*), 45  
 \_\_init\_\_() (*patito.duckdb.Relation method*), 53  
 \_\_iter\_\_() (*patito.duckdb.Relation method*), 85  
 \_\_len\_\_() (*patito.duckdb.Relation method*), 86  
 \_\_str\_\_() (*patito.duckdb.Relation method*), 86

## A

add\_prefix() (*patito.duckdb.Relation method*), 54  
 add\_suffix() (*patito.duckdb.Relation method*), 55  
 aggregate() (*patito.duckdb.Relation method*), 56  
 all() (*patito.duckdb.Relation method*), 57  
 as\_query() (*patito.Database method*), 44

## C

cache\_path() (*patito.Database.Query method*), 43  
 case() (*patito.duckdb.Relation method*), 58  
 cast() (*patito.DataFrame method*), 15  
 cast() (*patito.duckdb.Relation method*), 59  
 clear\_caches() (*patito.Database.Query method*), 43  
 coalesce() (*patito.duckdb.Relation method*), 60  
 columns (*patito.\_docs.Model property*), 24  
 columns (*patito.duckdb.Relation property*), 61  
 count() (*patito.duckdb.Relation method*), 61  
 create\_enum\_types() (*patito.duckdb.Database method*), 46  
 create\_table() (*patito.duckdb.Database method*), 47  
 create\_table() (*patito.duckdb.Relation method*), 62  
 create\_view() (*patito.duckdb.Database method*), 47  
 create\_view() (*patito.duckdb.Relation method*), 63

## D

Database (*class in patito*), 40  
 DatabaseQuery (*class in patito.database*), 42  
 DataFrame (*class in patito*), 15  
 DataFrame() (*patito.Model method*), 23  
 default() (*patito.duckdb.Database class method*), 47

defaults (*patito.\_docs.Model property*), 24  
 derive() (*patito.DataFrame method*), 16  
 distinct() (*patito.duckdb.Relation method*), 63  
 drop() (*patito.DataFrame method*), 17  
 drop() (*patito.duckdb.Relation method*), 64  
 drop() (*patito.Model class method*), 25  
 dtypes (*patito.\_docs.Model property*), 26

## E

empty\_relation() (*patito.duckdb.Database method*), 48  
 example() (*patito.Model class method*), 26  
 example\_value() (*patito.Model class method*), 27  
 examples() (*patito.Model class method*), 28  
 except\_() (*patito.duckdb.Relation method*), 64  
 execute() (*patito.duckdb.Database method*), 49  
 execute() (*patito.duckdb.Relation method*), 65

## F

Field (*class in patito*), 39  
 fill\_null() (*patito.DataFrame method*), 18  
 filter() (*patito.duckdb.Relation method*), 66  
 from\_connection() (*patito.duckdb.Database class method*), 50  
 from\_row() (*patito.Model class method*), 29

## G

get() (*patito.DataFrame method*), 19  
 get() (*patito.duckdb.Relation method*), 67

## I

inner\_join() (*patito.duckdb.Relation method*), 68  
 insert\_into() (*patito.duckdb.Relation method*), 69  
 intersect() (*patito.duckdb.Relation method*), 70

## J

join() (*patito.duckdb.Relation method*), 71  
 join() (*patito.Model class method*), 30

## L

LazyFrame() (*patito.Model method*), 24

`left_join()` (*patito.duckdb.Relation method*), 72  
`limit()` (*patito.duckdb.Relation method*), 73

## M

`Model` (*class in patito*), 23

## N

`non_nullable_columns` (*patito.\_docs.Model property*), 31  
`nullable_columns` (*patito.\_docs.Model property*), 31

## O

`order()` (*patito.duckdb.Relation method*), 74

## P

`pandas_examples()` (*patito.Model class method*), 32  
`prefix()` (*patito.Model class method*), 32

## Q

`query()` (*patito.Database method*), 44  
`query()` (*patito.duckdb.Database method*), 50  
`query_string()` (*patito.Database.Query method*), 43

## R

`read_csv()` (*patito.DataFrame class method*), 20  
`refresh_cache()` (*patito.Database.Query method*), 43  
`rename()` (*patito.duckdb.Relation method*), 75  
`rename()` (*patito.Model class method*), 33

## S

`select()` (*patito.duckdb.Relation method*), 76  
`select()` (*patito.Model class method*), 33  
`set_alias()` (*patito.duckdb.Relation method*), 77  
`set_model()` (*patito.DataFrame method*), 21  
`set_model()` (*patito.duckdb.Relation method*), 77  
`sql_types` (*patito.\_docs.Model property*), 34  
`suffix()` (*patito.Model class method*), 35

## T

`table()` (*patito.duckdb.Database method*), 51  
`to_df()` (*patito.duckdb.Relation method*), 78  
`to_pandas()` (*patito.duckdb.Relation method*), 79  
`to_relation()` (*patito.duckdb.Database method*), 52  
`to_series()` (*patito.duckdb.Relation method*), 79  
`types` (*patito.duckdb.Relation attribute*), 80

## U

`union()` (*patito.duckdb.Relation method*), 80  
`unique_columns` (*patito.\_docs.Model property*), 35

## V

`valid_dtypes` (*patito.\_docs.Model property*), 36

`valid_sql_types` (*patito.\_docs.Model property*), 36  
`validate()` (*patito.DataFrame method*), 22  
`validate()` (*patito.Model class method*), 37

## W

`with_columns()` (*patito.duckdb.Relation method*), 81  
`with_fields()` (*patito.Model class method*), 38  
`with_missing_defaultable_columns()` (*patito.duckdb.Relation method*), 82  
`with_missing_nullable_columns()` (*patito.duckdb.Relation method*), 83