

ducktide

Immutable Pydantic models and append-fast time series on DuckDB and Polars

Thomas Schmelzer

Development version after 0.1.0 September 2026

Abstract

ducktide is a small persistence layer for Python. It stores two kinds of data: *reference data* (instruments, sensors, accounts) as frozen Pydantic models in relational tables, and *time series* (prices, volumes, readings) as columnar tables written by idempotent bulk upserts and read back as Polars DataFrames. Both run on DuckDB, in memory or in a single file, with no server and no ORM framework. This document introduces the library, explains the design decisions behind it, and works through its API with examples. Every example was run against the development version that follows ducktide 0.1.0, and the output shown is what it printed.

Contents

1	Introduction	2
1.1	Installation	2
1.2	A first look	2
2	Design	3
2.1	Two stores, one engine	3
2.2	One model, one table	4
2.3	Values, not active records	5
2.4	Upsert semantics for time series	5
2.5	Physical layout	5
2.6	Safety	6
2.7	Errors	6
3	Entity tables in practice	6
3.1	Defining and inspecting a table	6
3.2	Writing	7
3.3	Reading	7
3.4	Values are immutable	8
3.5	A model with a non-default key and exact decimals	8
3.6	Frames, Parquet and CSV	9
4	Time series in practice	9
4.1	Ingest, correct, backfill	9
4.2	Reading	11
4.3	Other series keys and schemas	11
4.4	Time zones	12
4.5	Compaction, Parquet and CSV	12
4.6	Analysis with Polars	13

5	Bringing the two together	13
5.1	TimeSeriesModel	13
5.2	Many instruments, one write	14
5.3	Resampling	14
5.4	Selecting in one store, reading in the other	15
6	Deployment patterns	15
6.1	Files and read-only readers	15
6.2	A default database for notebooks and tests	16
6.3	Testing	16
7	Current limitations	16
8	API at a glance	17

1 Introduction

A great deal of analytical code has the same persistence needs. There is a modest set of *things*: financial instruments, sensors, customers. Each has a handful of attributes and changes rarely. And there is a large volume of *measurements* about those things, arriving continuously, occasionally late, sometimes corrected after the fact. A research notebook, a nightly ingest job and a backtest all want to read both, and none of them wants to run a database server.

DuckDB already solves the hard part. It is an embedded, columnar SQL engine that runs inside the Python process, stores a whole database in one file, reads and writes Parquet and CSV natively, and hands query results to Polars without copying them. `ducktide` does not replace any of that. It is candy on top of the DuckDB cake: a thin, opinionated layer that turns a few hundred lines of SQL you would otherwise write, and get subtly wrong, into a small API.

What the layer adds:

- **One definition per table.** A Pydantic model *is* the schema. Column names, DuckDB types and NOT NULL constraints are derived from its fields.
- **Typed values back.** Entity reads return validated, frozen model instances, never tuples.
- **Correct upserts.** Time-series ingestion merges on a series key, so re-ingesting overlapping data is safe and corrections and late rows land.
- **Performance traps avoided.** Bulk inserts go through one frame instead of row-by-row statements, fragmented frames are rechunked before ingest, and tables can be compacted by series key.
- **Safe SQL.** Identifiers are validated and quoted in one module, and values always travel as bound parameters.

When you need something the layer does not cover, the DuckDB connection is right there, and plain SQL works on the same tables.

1.1 Installation

```
| pip install ducktide          # or: uv add ducktide
```

`ducktide` requires Python 3.11 or later. Its runtime dependencies are DuckDB, Polars, Pydantic and PyArrow (used by DuckDB to hand results to Polars).

1.2 A first look

The shortest useful program defines a model, attaches it to a database and reads it back:

```

from ducktide import DB, DomainModel, Table

class Sensor(DomainModel):
    id: int
    name: str
    site: str

db = DB(tables_map={"sensor": Table.of(Sensor)}) # in memory
db.sensor.insert(Sensor(id=1, name="north", site="berlin"))
db.sensor.get(1)

```

```
| Sensor(id=1, name='north', site='berlin')
```

The time-series half is just as short. The table is created on the first ingest:

```

from datetime import datetime
import polars as pl
from ducktide import TimeSeriesDB

ts = TimeSeriesDB() # or
    TimeSeriesDB("prices.duckdb")
ts.ingest("prices", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 2), datetime(2025, 1, 3)],
    "instrument_id": [1, 1],
    "close": [100.0, 101.0],
}))
ts.get_timeseries_frame("prices", instrument_id=1) # a polars.DataFrame

```

2 Design

2.1 Two stores, one engine

ducktide has two entry points, `DB` and `TimeSeriesDB`. They are split by the *shape* of the data and by how it is read and written, not by storage engine: both are DuckDB. Table 1 contrasts them.

	Reference data (DB + Table)	Time series (TimeSeriesDB)
Examples	instruments, exchanges, sensors	prices, volumes, readings
Volume	hundreds to thousands of rows	millions of rows
Identity	a primary key	series key plus timestamp
Schema	declared by a Pydantic model	taken from the first frame ingested
Writes	inserts of validated objects	bulk upserts (MERGE)
Reads	frozen model instances	Polars DataFrames, sorted by time
Tuning	none needed	physical row order (compact)

Table 1: The two halves of ducktide.

Forcing both into one abstraction would hurt one of them. Row objects make time series slow: building a million Pydantic instances takes seconds and a great deal of memory for what Polars returns in milliseconds. Bare frames, in turn, would cost reference data its validation and types.

Keeping the two in separate files pays off operationally too:

- **Writers do not block each other.** DuckDB permits one read-write process per file. A nightly price ingest holding the time-series file does not lock the instrument table.
- **Different lifecycles.** Reference data is small and worth backing up or versioning. Time series are large, can usually be re-downloaded from a vendor, and can be rebuilt without

touching the definitions.

- **Read-only fan-out.** Many processes can open the time-series file with `read_only=True` while the reference database stays writable.

`TimeSeriesModel` connects the two: a model stored in DB knows its time-series table and its `instrument_id`, and can fetch or ingest its own series (Section 5). Figure 1 shows the pieces.

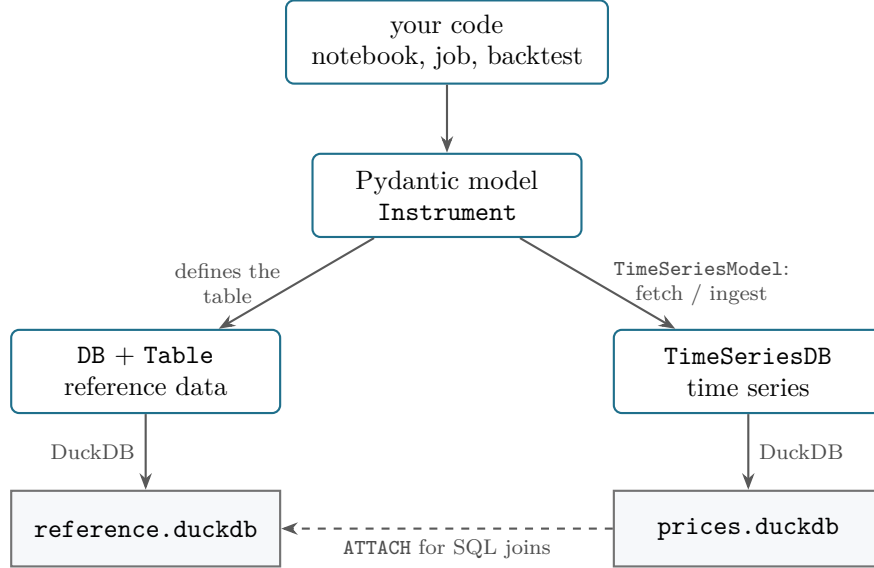


Figure 1: Architecture. One model describes a reference table and links to its series. Each store owns one DuckDB file.

2.2 One model, one table

A ducktide table is described by exactly one Pydantic model. Its fields *are* the columns, in declaration order. There is no second “ORM class” and no hand-written `CREATE TABLE` to keep in step with it. The mapping from Python types to DuckDB types is deliberately small and exact (Table 2). The lookup is on the exact type, so `bool` never falls through to `int` and `datetime` never to `date`.

Python	DuckDB	Python	DuckDB
<code>bool</code>	<code>BOOLEAN</code>	<code>date</code>	<code>DATE</code>
<code>int</code>	<code>BIGINT</code>	<code>datetime</code>	<code>TIMESTAMP</code>
<code>float</code>	<code>DOUBLE</code>	<code>time</code>	<code>TIME</code>
<code>str</code>	<code>VARCHAR</code>	<code>timedelta</code>	<code>INTERVAL</code>
<code>bytes</code>	<code>BLOB</code>	<code>UUID</code>	<code>UUID</code>
<code>Decimal</code>	<code>DECIMAL</code>		

Table 2: Type mapping used by `column_definitions`.

Three rules complete the derivation:

1. A field annotated `X | None` is nullable. Every other field is `NOT NULL`.
2. The primary-key field (default `id`) gets `PRIMARY KEY`.
3. `sql_types={column: definition}` replaces a column’s derived definition wholesale, constraints included. It covers anything the mapping cannot express: `UNIQUE`, a `DECIMAL` precision, or a type outside the table.

Column names are quoted in every statement ducktide builds, so a field may be named after a SQL keyword:

```
from datetime import datetime
from ducktide import DB, DomainModel, Table

class Event(DomainModel):
    id: int
    order: int
    at: datetime

db = DB(tables_map={"event": Table.of(Event)})
db.event.insert(Event(id=1, order=2, at=datetime(2025, 1, 2, 9, 30)))
db.event.select(order_after=1)
```

```
| [Event(id=1, order=2, at=datetime.datetime(2025, 1, 2, 9, 30))]
```

2.3 Values, not active records

Models carry no `save()`, `find()` or `delete()`. All persistence goes through a `Table`, the repository. Domain objects stay plain values that can be created in a test, passed between threads, hashed and compared, without dragging a connection along.

The optional base class `DomainModel` fixes the configuration ducktide recommends: `frozen=True`, so a row read back is never mutated behind the table's back, and stripping of surrounding whitespace from strings (`str_strip_whitespace=True`). Any Pydantic model can back a table; `DomainModel` is a convenience, not a requirement.

2.4 Upsert semantics for time series

A time-series row is identified by its *key*: the series columns plus the timestamp. By default the series column is `instrument_id` when the frame has one, and there are no series columns otherwise (a single series). `ingest` turns a frame into one DuckDB `MERGE` on that key:

- a row whose key is already stored **replaces** it (a correction);
- every other row is **inserted**, whatever its timestamp (a late row lands in the past rather than being dropped);
- within one frame, the **last row for a key wins**;
- a NULL key value matches a stored NULL instead of creating a duplicate;
- with `on_conflict="ignore"`, stored rows are kept and only new keys are added.

Ingest is therefore *idempotent*: re-running yesterday's job, or ingesting a window that overlaps what is stored, leaves the table correct. That property is what makes retries and backfills boring, which is the point.

2.5 Physical layout

DuckDB stores rows in blocks and keeps each block's minimum and maximum per column, so it can skip blocks that cannot match a filter. Ingestion appends in arrival order. When whole days arrive for every instrument at once, each instrument is spread across the whole table, and a read for one instrument has to touch nearly every block. `compact` rewrites a table sorted by series key, then timestamp. It is a trade-off, not a free speed-up (Table 3, one million daily bars for 500 instruments).

Read	time order	compacted
one instrument, one year	1.1 ms	0.75 ms
one day, all instruments	0.3 ms	1.4 ms
mean per instrument, whole table	0.65 ms	0.9 ms

Table 3: Effect of `compact` on read latency.

Compact only if reads for single series dominate. A table loaded one instrument at a time is already grouped and gains nothing. New ingests land unsorted again, so compact periodically, for example after each day’s ingest.

2.6 Safety

All SQL containing identifiers is composed in one module, `ducktide.utils.sql`. Table, schema and column names pass through `validate_identifier` and `quote_identifier` before they reach a statement. Keyword filters are checked against the table’s columns, and every data value is a bound parameter. The raw escape hatches (`select(where_clause, where_params)`, `TimeSeriesDB.query`) take `?` placeholders. Use them, and never format values into SQL strings.

2.7 Errors

Every ducktide exception derives from `DucktideError`:

```
DucktideError
+-- ValidationError      bad identifier, unknown filter, missing key column
+-- DataError           data processing or type problems
+-- DatabaseError       writes to a store opened read-only
    +-- DatabaseConnectionError
    +-- QueryError       constraint violations, failed time-series reads
```

Lookups that find nothing raise `KeyError`, as a mapping would. Reading a time-series table that does not exist returns an empty `DataFrame`: a missing table means “no data yet”, not an error.

3 Entity tables in practice

3.1 Defining and inspecting a table

`column_definitions` shows exactly what ducktide will create:

```
from datetime import date
from ducktide import DB, DomainModel, Table
from ducktide.model import column_definitions

class Sensor(DomainModel):
    id: int
    name: str
    site: str
    installed: date | None = None

column_definitions(Sensor)

{'id': 'BIGINT PRIMARY KEY', 'name': 'VARCHAR NOT NULL',
 'site': 'VARCHAR NOT NULL', 'installed': 'DATE'}
```

```
column_definitions(Sensor, sql_types={"name": "VARCHAR NOT NULL UNIQUE"})
```

```
{'id': 'BIGINT PRIMARY KEY', 'name': 'VARCHAR NOT NULL UNIQUE',
 'site': 'VARCHAR NOT NULL', 'installed': 'DATE'}
```

Tables are attached to a DB through `tables_map`. Each key becomes an attribute of the database, and `db.table` maps model classes to their tables:

```
db = DB(tables_map={"sensor": Table.of(Sensor)}) # or db_path="sensors.duckdb"
db.table[Sensor] is db.sensor
```

```
True
```

`Table.of` takes three options: `name=` (default: the lower-cased class name), `primary_key=` (default "id") and `sql_types=`. The table is created with `CREATE TABLE IF NOT EXISTS`, so opening an existing file is harmless.

3.2 Writing

```
db.sensor.insert(Sensor(id=1, name="north", site="berlin", installed=date(2024, 3,
1)))
db.sensor.bulk_insert([
    Sensor(id=2, name="south", site="zurich"),
    Sensor(id=3, name="east", site="berlin", installed=date(2025, 6, 15)),
])
len(db.sensor)
```

```
3
```

`insert(*objs)` takes one or more objects, and delegates to `bulk_insert` for more than one. `bulk_insert` accepts any iterable and hands DuckDB a single frame instead of running one statement per row. Use it for anything beyond a handful of objects. `DB.insert(*objs)` routes each object to its model's table, which is convenient when a batch mixes models.

Constraint violations surface as `QueryError`:

```
from ducktide.exceptions import QueryError

try:
    db.sensor.insert(Sensor(id=1, name="dup", site="x"))
except QueryError as exc:
    print(exc)
```

```
Constraint violation inserting into 'sensor': Constraint Error: Duplicate key "id:
1"
violates primary key constraint.
```

3.3 Reading

Point lookups by primary key or by any column:

```
db.sensor.get(2)
db.sensor[3].name
db.sensor.get_by("name", "east")
```

```
Sensor(id=2, name='south', site='zurich', installed=None)
'east'
Sensor(id=3, name='east', site='berlin', installed=datetime.date(2025, 6, 15))
```

All three raise `KeyError` when no row matches (`get(None)` returns `None`).

`select` filters with keyword arguments. A bare column name tests equality, and `None` tests for SQL `NULL`. The suffixes `_before` (`<`), `_after` (`>`), `_at_or_before` (`≤`) and `_at_or_after` (`≥`) express ranges. Several filters combine with `AND`:

```
[s.id for s in db.sensor.select(site="berlin")]
[s.id for s in db.sensor.select(installed=None)]
[s.id for s in db.sensor.select(installed_after=date(2025, 1, 1))]
[s.id for s in db.sensor.select(site="berlin", id_at_or_before=2)]
```

```
[1, 3]
[2]
[3]
[1]
```

Filter names are validated against the table, so a typo fails loudly:

```
db.sensor.select(colour="red")
```

```
ValidationError: Unknown filter 'colour' for table 'sensor'. Valid columns: id,
    name,
site, installed. Comparison suffixes: _after, _at_or_after, _at_or_before, _before
```

For anything keywords cannot express, pass a raw `WHERE` clause with placeholders. It is appended last, so `ORDER BY` and `LIMIT` work:

```
[s.name for s in db.sensor.select("site = ? ORDER BY name", ["berlin"])]
```

```
['east', 'north']
```

A table is also a small collection: `len(table)`, `bool(table)`, `iter(table)`, and the properties `exists` and `empty`.

3.4 Values are immutable

Rows come back frozen. To change one, derive a new value and write it:

```
s = db.sensor.get(1)
s.site = "paris" # raises pydantic ValidationError
moved = s.model_copy(update={"site": "paris"})
```

```
Sensor(id=1, name='north', site='paris', installed=datetime.date(2024, 3, 1))
```

Table has no update or delete method yet. Both are one statement through the connection:

```
db.execute_query("UPDATE sensor SET site = ? WHERE id = ?", ["paris", 1])
db.execute_query("DELETE FROM sensor WHERE id = ?", [2])
[(x.id, x.site) for x in db.sensor]
```

```
[(1, 'paris'), (3, 'berlin')]
```

`Table.execute(sql, params)` runs a query and reads the rows back as models, which is the way to use SQL that `select` cannot express while still getting typed values:

```
db.sensor.execute("SELECT * FROM sensor WHERE site = ?", ["paris"])
```

```
[Sensor(id=1, name='north', site='paris', installed=datetime.date(2024, 3, 1))]
```

3.5 A model with a non-default key and exact decimals

```

from datetime import datetime
from decimal import Decimal

class Trade(DomainModel):
    trade_id: int
    ticker: str
    qty: int
    price: Decimal
    executed_at: datetime

column_definitions(Trade)          # fails: there is no field called id

```

```
| ValidationError: Trade has no field(s) id
```

```

trades = Table.of(Trade, name="trades", primary_key="trade_id",
                  sql_types={"price": "DECIMAL(18, 6) NOT NULL"})
db2 = DB(tables_map={"trades": trades})
db2.trades.insert(Trade(trade_id=7, ticker="ACME", qty=100,
                        price=Decimal("101.25"), executed_at=datetime(2025, 1, 2,
                            14, 30)))
db2.trades.get_by("trade_id", 7)

```

```

| Trade(trade_id=7, ticker='ACME', qty=100, price=Decimal('101.250000'),
|   executed_at=datetime.datetime(2025, 1, 2, 14, 30))

```

Note that `get` looks up the configured primary key, so `db2.trades.get(7)` works too.

3.6 Frames, Parquet and CSV

```
db.sensor.to_frame()
```

```

| id | name | site | installed |
|----|-----|-----|-----|
| 1  | north | berlin | 2024-03-01 |
| 2  | south | zurich | null      |
| 3  | east  | berlin | 2025-06-15 |

```

`to_parquet(path)` and `to_csv(path, delimiter=",", header=True, overwrite=True)` export through DuckDB's native writers. `from_parquet(path)` and `from_csv(path)` append a file's rows to the table and return how many they imported. The file must match the table's columns. Paths are validated before they reach SQL, and an export creates any missing parent folders.

```

db.sensor.to_parquet("export/2025/sensors.parquet")    # missing folders are created

fresh = DB(tables_map={"sensor": Table.of(Sensor)})
fresh.sensor.from_parquet("export/2025/sensors.parquet")

```

```
| 3
```

4 Time series in practice

4.1 Ingest, correct, backfill

```
from datetime import date, datetime
```

```
import polars as pl
from ducktide import TimeSeriesDB

ts = TimeSeriesDB()
ts.ingest("prices", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 2), datetime(2025, 1, 3)] * 2,
    "instrument_id": [1, 1, 2, 2],
    "close": [100.0, 101.0, 50.0, 49.5],
}))
ts.tables(), ts.has_table("prices")
```

```
|(['prices'], True)
```

The table was created from the frame's schema. The next frame carries a correction for 3 January and a late bar from the previous year. Both land:

```
ts.ingest("prices", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 3), datetime(2024, 12, 31)],
    "instrument_id": [1, 1],
    "close": [101.2, 99.0],
}))
ts.get_timeseries_frame("prices", instrument_id=1)
```

timestamp	instrument_id	close
2024-12-31 00:00:00	1	99.0
2025-01-02 00:00:00	1	100.0
2025-01-03 00:00:00	1	101.2

With `on_conflict="ignore"`, stored rows win and only new keys are added. That suits a source you trust less than what is already stored:

```
ts.ingest("prices", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 3), datetime(2025, 1, 6)],
    "instrument_id": [1, 1],
    "close": [0.0, 102.0],
}), on_conflict="ignore")
ts.get_timeseries_frame("prices", instrument_id=1)["close"].to_list()
```

```
| [99.0, 100.0, 101.2, 102.0]
```

Duplicate keys inside one frame collapse to the last row:

```
ts.ingest("prices", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 7)] * 2,
    "instrument_id": [2, 2],
    "close": [48.0, 48.1],
}))
ts.get_timeseries_frame("prices", instrument_id=2)["close"].to_list()
```

```
| [50.0, 49.5, 48.1]
```

A frame without the timestamp or key columns is rejected before anything is written:

```
ts.ingest("prices", pl.DataFrame({"close": [1.0]}))
```

```
| ValidationError: frame is missing key column(s): timestamp
```

4.2 Reading

`get_timeseries_frame` takes the table and the optional filters `instrument_id`, `start`, `end` and `timezone`. It returns a frame sorted by time, flagged as sorted for Polars. Both bounds are inclusive. A `datetime` bound is used to the instant; a `date` bound covers the whole day, so `start=date(...)` begins at that midnight and `end=date(...)` runs up to the next one. A missing table yields an empty frame.

```
ts.get_timeseries_frame("prices", start=date(2025, 1, 3), end=date(2025, 1, 6)).height
ts.get_timeseries_frame("missing").shape
```

```
3
(0, 0)
```

With intraday data the difference between the two kinds of bound shows:

```
bars = TimeSeriesDB()
bars.ingest("m", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 2, 9, 30), datetime(2025, 1, 2, 16, 0),
datetime(2025, 1, 3)],
    "instrument_id": [1, 1, 1],
    "close": [1.0, 2.0, 3.0],
}))
bars.get_timeseries_frame("m", end=date(2025, 1, 2))["close"].to_list()
bars.get_timeseries_frame("m", end=datetime(2025, 1, 2, 9, 30))["close"].to_list()
```

```
[1.0, 2.0]
[1.0]
```

For SQL beyond these filters, `query(sql, *args)` returns a Polars frame. Values are passed positionally for the `?` placeholders:

```
ts.query("""
    SELECT instrument_id, avg(close) AS mean_close, count(*) AS n
    FROM prices GROUP BY instrument_id ORDER BY instrument_id
""")
ts.query("SELECT * FROM prices WHERE close > ? ORDER BY timestamp", 100.0).height
```

```
| instrument_id | mean_close | n |
|-----|-----|---|
| 1           | 100.55     | 4 |
| 2           | 49.2       | 3 |
2
```

4.3 Other series keys and schemas

Series need not be keyed by `instrument_id`. Pass `key=` with the columns identifying one series (the timestamp is added automatically). Table names may be schema-qualified. A missing schema is created on first ingest.

```
fx = pl.DataFrame({
    "timestamp": [datetime(2025, 1, 2)] * 2,
    "base": ["EUR", "GBP"], "quote": ["USD", "USD"], "rate": [1.035, 1.24],
})
ts.ingest("market.fx", fx, key=["base", "quote"])
ts.ingest("market.fx", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 2)], "base": ["EUR"], "quote": ["USD"], "rate":
[1.036],
```

```
}), key=["base", "quote"])
ts.query("SELECT base, quote, rate FROM market.fx ORDER BY base")
```

base	quote	rate
EUR	USD	1.036
GBP	USD	1.24

The timestamp column is called `timestamp` unless configured otherwise:

```
sensors = TimeSeriesDB(time_col="ts")
sensors.ingest("readings", pl.DataFrame({
    "ts": [datetime(2025, 1, 1, 0, 0), datetime(2025, 1, 1, 0, 5)],
    "sensor_id": [3, 3],
    "celsius": [4.1, 4.3],
}), key=["sensor_id"])
sensors.get_timeseries_frame("readings")
```

ts	sensor_id	celsius
2025-01-01 00:00:00	3	4.1
2025-01-01 00:05:00	3	4.3

4.4 Time zones

Timezone-aware timestamps come back in UTC, whatever the time zone of the machine running the code: `TimeSeriesDB` pins its DuckDB session to UTC. Pass `timezone=` to convert on the way out:

```
tz = TimeSeriesDB()
tz.ingest("ticks", pl.DataFrame({
    "timestamp": [datetime(2025, 1, 2, 14, 30)], "px": [1.0],
}).with_columns(pl.col("timestamp").dt.replace_time_zone("UTC")))
tz.get_timeseries_frame("ticks", timezone="America/New_York")
```

timestamp	px
2025-01-02 09:30:00 EST	1.0

4.5 Compaction, Parquet and CSV

```
ts.compact("prices") # grouped by instrument_id, then time
ts.compact("market.fx", key=["base", "quote"])

ts.export_parquet("prices", "prices.parquet")
archive = TimeSeriesDB()
archive.import_parquet("prices.parquet", "prices")
archive.get_timeseries_frame("prices").height
```

7

`export_csv` and `import_csv` work the same way. `TimeSeriesDB` is a context manager that closes its connection on exit:

```
with TimeSeriesDB("prices.duckdb") as store:
    latest = store.query("SELECT max(timestamp) AS t FROM prices")
```

4.6 Analysis with Polars

The frame that comes back is ordinary Polars, so analysis starts immediately:

```
daily = TimeSeriesDB()
daily.ingest("daily", pl.DataFrame({
    "timestamp": [datetime(2025, 1, d) for d in (2, 3, 6, 7, 8)],
    "instrument_id": [1] * 5,
    "close": [100.0, 102.0, 101.0, 104.0, 103.0],
}))
daily.get_timeseries_frame("daily", instrument_id=1).with_columns(
    ret=pl.col("close").pct_change(),
    ma3=pl.col("close").rolling_mean(3),
).select("timestamp", "close", "ret", "ma3")
```

timestamp	close	ret	ma3
2025-01-02 00:00:00	100.0	null	null
2025-01-03 00:00:00	102.0	0.02	null
2025-01-06 00:00:00	101.0	-0.009804	101.0
2025-01-07 00:00:00	104.0	0.029703	102.333333
2025-01-08 00:00:00	103.0	-0.009615	102.666667

5 Bringing the two together

5.1 TimeSeriesModel

A model that mixes in `TimeSeriesModel` declares its time-series table as a class variable and exposes an `instrument_id`. It then fetches and ingests its own series. The repository is passed in, not stored, so the model stays a plain value.

```
import tempfile
from pathlib import Path
from typing import ClassVar
import polars as pl
from ducktape import DB, DomainModel, Table, TimeSeriesDB, TimeSeriesModel

class Instrument(DomainModel, TimeSeriesModel):
    table_name: ClassVar[str] = "bars"

    id: int
    ticker: str
    exchange: str

    @property
    def instrument_id(self) -> int:
        return self.id

root = Path(tempfile.mkdtemp())
ref = DB(tables_map={"instrument": Table.of(Instrument)}, db_path=root /
    "reference.duckdb")
ts = TimeSeriesDB(root / "bars.duckdb")

ref.instrument.bulk_insert([
    Instrument(id=1, ticker="ACME", exchange="XNYS"),
    Instrument(id=2, ticker="GLOBEX", exchange="XNAS"),
    Instrument(id=3, ticker="INITECH", exchange="XNYS"),
])
```

`TimeSeriesModel.ingest` expects vendor-style frames with an ISO-8601 `ts_event` column. It parses that into a UTC timestamp, adds the model's `instrument_id`, and ingests into `table_name`. The original `ts_event` column is kept.

```
def bars(px0: float) -> pl.DataFrame:
    """Four 15-minute OHLCV bars starting at px0."""
    stamps = [f"2025-01-02T{h}:{m}:00.000000+0000" for h in ("14", "15") for m in ("30", "45")]
    px = [px0 + i for i in range(4)]
    return pl.DataFrame({
        "ts_event": stamps,
        "open": px,
        "high": [p + 0.5 for p in px],
        "low": [p - 0.5 for p in px],
        "close": [p + 0.25 for p in px],
        "volume": [100, 200, 150, 50],
    })

acme = ref.instrument.get(1)
acme.ingest(ts, bars(100.0))
acme.get_timeseries_frame(ts).select("timestamp", "instrument_id", "close")
```

timestamp	instrument_id	close
2025-01-02 14:30:00 UTC	1	100.25
2025-01-02 14:45:00 UTC	1	101.25
2025-01-02 15:30:00 UTC	1	102.25
2025-01-02 15:45:00 UTC	1	103.25

5.2 Many instruments, one write

A write to DuckDB costs roughly the same for one row as for thousands. When a day of bars arrives for hundreds of instruments, ingest them together. `ingest_many` groups the pairs by table and issues one write per table, with the same result as calling `ingest` for each pair in order:

```
Instrument.ingest_many(ts, [(i, bars(10.0 * i.id)) for i in ref.instrument if i.id != 1])
ts.query("SELECT instrument_id, count(*) AS n, max(close) AS last FROM bars GROUP BY 1 ORDER BY 1")
```

instrument_id	n	last
1	4	103.25
2	4	23.25
3	4	33.25

5.3 Resampling

`get_timeseries_frame(repo, every=...)` resamples OHLCV bars with the usual aggregation (first open, max high, min low, last close, summed volume). The frame must have those five columns and a `timestamp` column.

```
acme.get_timeseries_frame(ts, every="1h")
```

timestamp	open	high	low	close	volume
2025-01-02 14:00:00 UTC	100.0	101.5	99.5	101.25	300
2025-01-02 15:00:00 UTC	102.0	103.5	101.5	103.25	200

5.4 Selecting in one store, reading in the other

The natural pattern: choose the instruments with a typed query against reference data, then read their series.

```
nyse = ref.instrument.select(exchange="XNYS")
frames = [i.get_timeseries_frame(ts).with_columns(ticker=pl.lit(i.ticker)) for i in
           nyse]
pl.concat(frames).group_by("ticker").agg(pl.col("volume").sum()).sort("ticker")
```

ticker	volume
ACME	500
INITECH	500

When one SQL statement across both stores is the better tool, attach the reference file to the time-series connection. The raw connection is `ts.con` (and `db.connection` on a DB). A file can only be open read-write in one place, so close the DB first, or open it read-only.

```
ref.close(); ts.close()
ts = TimeSeriesDB(root / "bars.duckdb")
ts.con.execute(f"ATTACH '{root / 'reference.duckdb'}' AS ref (READ_ONLY)")
ts.query("""
    SELECT i.exchange, sum(b.volume)::BIGINT AS volume
    FROM bars AS b JOIN ref.instrument AS i ON i.id = b.instrument_id
    GROUP BY i.exchange ORDER BY i.exchange
""")
```

exchange	volume
XNAS	500
XNYS	1000

6 Deployment patterns

6.1 Files and read-only readers

Both stores run in memory by default. Pass a path to persist: `db_path=` for a DB, the first argument for a `TimeSeriesDB`. A typical layout is one writer job per file and any number of read-only readers:

```
# ingest job (the only writer)
with TimeSeriesDB("prices.duckdb") as ts:
    ts.ingest("prices", todays_bars)
    ts.compact("prices")

# research notebooks, dashboards, backtests (as many as you like)
reader = TimeSeriesDB("prices.duckdb", read_only=True)
reader.get_timeseries_frame("prices", instrument_id=42)
```

A read-only store refuses every write (insert, bulk insert, ingest, compact, imports, dropping tables) with a `DatabaseError` before anything reaches DuckDB. Exports still work, since they write a file rather than the database:

```
reader.ingest("prices", todays_bars)
```

```
DatabaseError: The time-series database is read-only: ingest needs a writable
connection
```

6.2 A default database for notebooks and tests

`ducktide.context` provides a `ContextVar`-scoped default database, so helper functions need not take a `db` argument. It is safe across threads and asyncio tasks.

```
from ducktide.context import use_db, get_default_db

def exchanges() -> list[str]:
    return sorted({i.exchange for i in get_default_db().instrument})

with use_db(ref):
    exchanges()
```

```
| ['XNAS', 'XNYS']
```

`set_default_db` and `clear_default_db` do the same without a `with` block.

6.3 Testing

Because everything runs in memory by default, a test fixture is one line and needs no cleanup:

```
import pytest

@pytest.fixture
def db():
    return DB(tables_map={"instrument": Table.of(Instrument)})

@pytest.fixture
def ts():
    return TimeSeriesDB()

def test_correction_replaces_bar(ts):
    bar = {"timestamp": [datetime(2025, 1, 2)], "instrument_id": [1]}
    ts.ingest("prices", pl.DataFrame(**bar, "close": [1.0]))
    ts.ingest("prices", pl.DataFrame(**bar, "close": [2.0]))
    assert ts.get_timeseries_frame("prices")["close"].to_list() == [2.0]
```

7 Current limitations

- **Table names that are SQL keywords.** Column names are quoted, table names are not yet. A model class named after a keyword (`class Order` gives the table `order`) fails at table creation with a DuckDB parser error. Name the table explicitly: `Table.of(Order, name="orders")`.
- **No update or delete on Table.** Use `db.execute_query` with placeholders, as shown above.
- **One writer per file.** This is DuckDB's rule, not ducktide's. Plan processes around it: one ingest job per time-series file, read-only readers everywhere else.
- **compact is not free.** It helps single-series reads and slows cross-sectional ones. DuckDB does not shrink the file afterwards.
- **Resampling assumes OHLCV.** `every=` aggregates open/high/low/close/volume on timestamp. For other shapes, resample the returned frame with Polars' `group_by_dynamic` directly.
- **Raw SQL is raw.** `execute_query` and `query` pass DuckDB's own exceptions through, including on a read-only store.

8 API at a glance

DB (tables_map, db_path=":memory:", read_only=False)	
db.<name>, db.table[Model]	the attached tables
insert(*objs)	route objects to their tables
execute_query(sql, params)	raw SQL on the connection
close(), commit(), cursor(), drop_all_tables()	connection housekeeping
Table .of(Model, name=, primary_key="id", sql_types=)	
insert(*objs), bulk_insert(iterable)	write validated objects
get(id), table[id], get_by(col, value)	point lookups (KeyError if absent)
select(where_clause=None, where_params=None, **filters)	filtered reads
execute(sql, params)	raw SQL read back as models
to_frame(), to_parquet, to_csv, from_parquet, from_csv	frames and files
len, bool, iter, exists, empty	collection protocol
TimeSeriesDB (path=None, time_col="timestamp", read_only=False)	
ingest(table, frame, key=None, on_conflict="update")	idempotent upsert
get_timeseries_frame(table, instrument_id, start, end, timezone)	sorted reads
query(sql, *args)	raw SQL to a Polars frame
compact(table, key=None)	regroup rows by series
import_parquet, export_parquet, import_csv, export_csv	files
tables(), has_table(t), con, close()	catalog and connection
TimeSeriesModel (table_name: ClassVar[str], instrument_id property)	
get_timeseries_frame(repo, start, end, timezone, every)	this instrument's series
ingest(repo, frame), Model.ingest_many(repo, pairs)	write from ts_event frames

ducktide is MIT-licensed. Source, issues and the companion book: <https://github.com/Jebel-Quant/ducktide>.