TinyFlux

Release 0.1

Justin Fung

Nov 28, 2022

BASICS

1 Introduction 3

2 Installing TinyFlux 7

3 Getting Started 9

4 Preparing Data 13

5 Writing Data 15

6 Querying Data 17

7 Exploring Data 23

8 Updating Points 27

9 Removing Points 29

10 Working with Measurements 31

11 Timezones in TinyFlux 33

12 Tips for TinyFlux 37

13 Elements of Data in TinyFlux 39

14 TinyFlux Design Principles 43

15 TinyFlux Internals 45

16 TinyFlux API 47

17 Philosophy 69

18 Guidelines 71

19 Tooling and Conventions 73

20 Changelog 77

Python Module Index 79

Index 81

The tiny time series database, optimized for your happiness.

CHAPTER

ONE

INTRODUCTION

TinyFlux combines the simplicity of the document-oriented TinyDB with the concepts and design of the fully-fledged time series database known as InfluxDB.

TinyFlux is a pure Python module that supports database-like operations on an in-memory or file datastore. It is optimized for time series data and as such, is considered a "time series database" (or "tsdb" for short). It is not, however, a database server that supports traditional RDMS features like the management of concurrent connections, management of indexes in background processes, or the provisioning of access control. Before using TinyFlux, you should be sure that TinyFlux is right for your intended use-case.

1.1 Why Should I Use TinyFlux?

In TinyFlux, time comes first.

- Time in TinyFlux is a first-class citizen. TinyFlux expects and handles Python datetime objects with ease. Queries are optimized for time, above all else.

TinyFlux is a real time series database.

- Concepts around TinyFlux are based on InfluxDB. If you are looking for a gradual introduction into the world of time series databases, this is a great starting point. If your workflow outgrows the offerings of TinyFlux, you can jump to InfluxDB with very little introduction needed.

TinyFlux is written in pure, standard library Python.

- TinyFlux needs neither an external server nor any dependencies and works on all modern versions of Python.

TinyFlux is optimized for your happiness.

- Like TinyDB, TinyFlux is designed to be simple and easy to use by providing a straight-forward and clean API.

TinyFlux is tiny.

- The current source code has 2000 lines of code (with about 50% documentation) and 2000 lines of tests.

TinyFlux has 100% test coverage.

- No explanation needed.

If you have a moderate amount of time series data without the need or desire to provision and manage a full-fledged server and its configuration, and you want to interface easily with the greater Python ecosystem, TinyFlux might be the right choice for you.

1.2 When To Look at Other Options

You should not use TinyFlux if you need advanced database features like:

- access from multiple processes or threads
- an HTTP server
- management of relationships between tables
- access-control and users
- ACID guarantees
- high performance as the size of your dataset grows

If you have a large amount of data or you need advanced features and high performance, consider using databases like SQLite or InfluxDB.

1.3 What's the difference between TinyFlux and TinyDB?

At its core, TinyFlux is a *time series database* while TinyDB is a *document-oriented database*.

Let's break this down:

In TinyFlux, time is a "first-class citizen".

- In TinyDB, there is no special handling of time.

A TinyFlux database expects Python datetime objects to be passed with each and every data point.

- TinyDB does not accept datetime objects directly. In TinyDB, attributes representing time must be serialized and deserialized by the user, or an extension must added onto TinyDB to handle datetime objects.

In TinyFlux, queries are optimized for time.

- TinyFlux builds a small index in memory which includes an index on timestamps. This provides for ultra-fast search and retrieval of data when queries are time-based. TinyDB has no special mechanism for querying attributes of different types.

Data in TinyFlux is written to disk in "append-only" fashion.

- Irrespective of the current size of the database, inserting is always a constant-time operation on the order of nanoseconds. TinyFlux is optimized for time series datasets which are often write-heavy, as opposed to document-stores which are traditionally read-heavy. This allows high-frequency signals to be easily handled by TinyFlux. TinyDB does not expect high-frequency writes, and since it reads all data into memory before adding new data, its insert time increases linearly with the size of the database.

TinyFlux and TinyDB are both "schemaless".

- This means that attributes and their existence between items may differ with no exceptions being raised. TinyDB, as a document store, supports the storage of complex types including containers like arrays/lists and objects/dictionaries. TinyFlux, however, provides for just three types of attributes- numeric, string, and of course, datetime.

1.3.1 Got it, so should I use TinyFlux or TinyDB?

You should use TinyFlux if:

- Your data is naturally time series in nature. That is, you have many observations of some phenomenon over time with varying measurements. Examples include stock prices, daily temperatures, or the accelerometer readings on a running watch.
- You will be writing to the database at a regular, high frequency.

You should use TinyDB if:

- Your data has no time dimension. Examples include a database acting as a phonebook for Chicago, the catalogue of Beatles music, or configuration values for your dashboard app.
- You will be writing to the database infrequently.

CHAPTER

TWO

INSTALLING TINYFLUX

To install TinyFlux from PyPI, run:

```
$ pip install tinyflux
```

The latest development version is hosted on GitHub. After downloading, install using:

```
$ pip install .
```

CHAPTER

THREE

GETTING STARTED

Initialize a new TinyFlux database (or connect to an existing file store) with the following:

```
>>> from tinyflux import TinyFlux
>>> db = TinyFlux('db.csv')
```

`db` is now a reference to the TinyFlux database that stores its data in a file called `db.csv`.

An individual instance of data in a TinyFlux database is known as a "Point". In a traditional relational database, this would called called a "row", and in a document-oriented database it is called a "document". A TinyFlux `Point` is a convenient object for storing its four main attributes:

Attribute	Python Type	Example
`measurement`	`str`	`"california air quality"`
`time`	`datetime`	`datetime.now(timezone.utc)`
`tags`	`Dict` of `str` keys and `str` values	`{"city": "Los Angeles", "parameter": "PM2.5"}`
`fields`	`Dict` of `str` keys and `float` or `int` values	`{"aqi": 112.0}`

In keeping with the analogy of a traditional RDMS, a `measurement` is like a table.

`time` is a field with the requirement that it is a `datetime` type, `tags` is a collection of string attributes, and `fields` is a collection of numeric attributes. TinyFlux is "schemaless", so tags and fields can be added/removed to any Point.

To make a Point, import the Point definition and annotate the Point with the desired attributes. If `measurement` is not defined, it takes the default table name of `_default`.

```
>>> from tinyflux import Point
>>> p1 = Point(
...     time=datetime.fromisoformat("2020-08-28T00:00:00-07:00"),
...     tags={"city": "LA"},
...     fields={"aqi": 112}
... )
>>> p2 = Point(
...     time=datetime.fromisoformat("2020-12-05T00:00:00-08:00"),
...     tags={"city": "SF"},
...     fields={"aqi": 128}
... )
```

To write to TinyFlux, simply:

```
>>> db.insert(p1)
>>> db.insert(p2)
```

All points can be retrieved from the database with the following:

```
>>> db.all()
[Point(time=2020-01-01T00:08:00-00:00, measurement=_default, tags=city:LA,␣
↪fields=aqi:112), Point(time=2020-12-05T00:08:00-00:00, measurement=_default,␣
↪tags=city:SF, fields=aqi:128)]
```

Note: TinyFlux will convert all time to UTC. Read more about it here: *Timezones in TinyFlux*.

TinyFlux also allows iteration over stored Points:

```
>>> for point in db:
>>>     print(point)
Point(time=2020-08-28T00:07:00-00:00, measurement=_default, tags=city:LA, fields=aqi:112)
Point(time=2020-12-05T00:08:00-00:00, measurement=_default, tags=city:SF, fields=aqi:128)
```

To query for Points, there are four query types- one for each of a Point's four attributes.

```
>>> from tinyflux import FieldQuery, MeasurementQuery, TagQuery, TimeQuery
>>> Time = TimeQuery()
>>> db.search(Time < datetime.fromisoformat("2020-11-00T00:00:00-08:00"))
[Point(time=2020-08-28T00:07:00-00:00, measurement=_default, tags=city:LA,␣
↪fields=aqi:112)]
>>> Field = FieldQuery()
>>> db.search(Field.aqi > 120)
[Point(time=2020-12-05T00:08:00-00:00, measurement=_default, tags=city:SF,␣
↪fields=aqi:128)]
>>> Tag = TagQuery()
>>> db.search(Tag.city == "LA")
[Point(time=2020-08-28T00:07:00-00:00, measurement=_default, tags=city:LA,␣
↪fields=aqi:112)]
>>> Measurement = MeasurementQuery()
>>> db.count(Measurement == "_default")
2
```

Points can also be updated:

```
>>> # Update the ``aqi`` field of the Los Angeles point.
>>> db.update(tag.city == "LA", tags={"aqi": 118})
>>> for point in db:
>>>     print(point)
Point(time=2020-08-28T00:07:00-00:00, measurement=_default, tags=city:LA, fields=aqi:118)
Point(time=2020-12-05T00:08:00-00:00, measurement=_default, tags=city:SF, fields=aqi:128)
```

Points can also be removed:

```
>>> db.remove(tag.city == "SF")
1
```

(continues on next page)

(continued from previous page)

```
>>> db.all()
[Point(time=2020-01-01T00:08:00-00:00, measurement=_default, tags=city:LA,␣
↪fields=aqi:112)]
```

Here is the basic syntax covered in this section:

Initialize a new TinyFlux Database	
`db = TinyFlux("my_db.csv")`	Initialize or connect to existing with `TinyFlux()`
Creating New Points	
`Point(...)`	Initialize a new point.
Inserting Points Into the Database	
`db.insert()`	Insert a point.
Retrieving Points	
`db.all()`	Get all points
`iter(db)`	Iterate over all points
`db.search(query)`	Get a list of points matching the query
`db.count(query)`	Count the number of points matching the query
Updating Points	
`db.update(query, ...)`	Update all points matching the query
Removing Points	
`db.remove(query)`	Remove all points matching the query
`db.remove_all()`	Remove all points
Querying TinyFlux	
`TimeQuery()`	Create a new time query object
`FieldQuery().f_key == 2`	Match any point that has a field `f_key` with value `== 2` (also possible: `!=`, `>`, `>=`, `<`, `<=`)

To continue with the introduction to TinyFlux, proceed to the next section, *Preparing Data*.

CHAPTER

FOUR

PREPARING DATA

Before inserting data into TinyFlux, data must be cast into specific types of objects known as a "Points". Here's an example:

```
>>> from tinyflux import Point
>>> from datetime import datetime, timezone
>>> p = Point(
...     measurement="city temperatures",
...     time=datetime(2022, 1, 1, tzinfo=timezone.utc),
...     tags={"city": "Greenwich", "country": "England"},
...     fields={"high": 52.0, "low": 41.0}
... )
```

This term "Point" comes from InfluxDB. A well-formed Point consists of four attributes:

- `measurement`: Known as a "table" in relational databases, its value type is `str`.
- `time`: The timestamp of the observation, its value is a Python `datetime` object that should be "timezone aware".
- `tags`: Text attributes of the observation as a Python `dict` of `str|str` key value pairs.
- `fields`: Numeric attributes of the observation as a Python `dict` of `str|int` or `str|float` key value pairs.

None of the four attributes is required during initialization; an empty Point can be initialized like the following:

```
>>> from tinyflux import Point
>>> Point()
Point(time=None, measurement=_default)
```

Notice that the `time` attribute is `None`, and the `measurement` attribute has taken the value of `_default`. The point also has no tags or fields. Tags and fields are not required, but from a user's perspective, such a data point has little meaning.

Note: Points that do not have `time` values take on timestamps *when they are inserted into TinyFlux, not when they are created*. If you want time to reflect the time of creation, set time like: `time=datetime.now(timezone.utc)`.

A default `measurement` is assigned to Points that are initialized without one.

Tags are string/string key value pairs. The reason for having separate attributes for `tags` and `fields` in TinyFlux (and in InfluxDB) is twofold: It enforces consistency of types and data on the user's side, and it allows the database to efficiently index on tags, which are attributes with low cardinality (compared to fields, which tend to have much higher variation across values).

Note: While both TinyDB and TinyFlux are "schemaless", TinyFlux does not support complex types as values. If you want to store documents, which are often collections rather than primitive types, take a look at TinyDB.

Hint: TinyFlux will raise a `ValueError` if you try to initialize a `Point` with incorrect types, so you can be sure you are not inserting malformed data into the database.

CHAPTER

FIVE

WRITING DATA

The standard method for inserting a new data point is through the `db.insert(...)` method. To insert more than one Point at the same time, use the `db.insert_multiple([...])` method, which accepts a `list` of points. This might be useful when creating a TinyFlux database from a CSV of existing observations.

Note: TinyFlux vs. TinyDB Alert!

In TinyDB there is a serious performance reason to use `db.insert_multiple([...])` over `db.insert(...)` as every write in TinyDB is preceeded by a full read of the data. TinyFlux inserts are *append-only* and are **not** preceeded by a read. Therefore, there is no significant *performance* reason to use `db.insert_multiple([...])` instead of `db.insert(...)`. If you are using TinyFlux to capture real-time data, you should insert points into TinyFlux as you see them, with `db.insert(...)`.

Example:

```
>>> from tinyflux import Point
>>> p = Point(
...     measurement="air quality",
...     time=datetime.fromisoformat("2020-08-28T00:00:00-07:00"),
...     tags={"city": "LA"},
...     fields={"aqi": 112}
... )
>>> db.insert(p)
```

To recap, these are the two methods supporting the insertion of data.

Methods	
`db.insert(point)`	Insert one Point into the database.
`db.insert_multiple([point, ...])`	Insert multiple Points into the database.

CHAPTER

SIX

QUERYING DATA

TinyFlux's query syntax will be familiar to users of popular ORM tools. It is similar to that of TinyDB, but TinyFlux contains four different query types, one for each of a point's four attributes.

The query types are:

- `TimeQuery` for querying points by `time`.
- `MeasurementQuery` for querying points by `measurement`.
- `TagQuery` for querying points by `tags`.
- `FieldQuery` for querying points by `fields`.

For the remainder of this section, query examples will be illustrated with the `.search()` method of a TinyFlux database. This is the most common way to query TinyFlux, and the method accepts a query and returns a `list` of `Point` objects matching the query. In addition, there are a handful of other database methods that take queries as argument and perform some sort of search. See the *Exploring Data* section for details.

Note: `.search()` will return Points in sorted time order by default. To return points in insertion order, pass the `sorted=False` argument, like: `db.search(query, sorted=False)`.

6.1 Simple Queries

Examples of the four basic query types are below:

6.1.1 Measurement Queries

To query for a specific measurement, the right-hand side of the `MeasurementQuery` should be a Python `str`:

```
>>> from tinydb import MeasurementQuery
>>> Measurement = MeasurementQuery()
>>> db.search(Measurement == "city temperatures")
```

6.1.2 Tag Queries

To query for tags, the *tag key* of interest takes the form of a query attribute (following the `.`), while the *tag value* forms the right-hand side. An example to illustrate:

```
>>> from tinydb import TagQuery
>>> Tags = TagQuery()
>>> db.search(Tags.city == "Greenwich")
```

This will query the database for all points with the tag key of `city` mapping to the tag value of `Greenwich`.

6.1.3 Field Queries

Similar to tags, to query for fields, the field key takes the form of a query attribute, while the field value forms the right-hand side:

```
>>> from tinydb import FieldQuery
>>> Fields = FieldQuery()
>>> db.search(Fields.high > 50.0)
```

This will query the database for all points with the field key of `high` exceeding the value of 50.0.

Some tag keys and field keys are not valid Python identifiers (for example, if the key contains whitespace), and can alternately be queried with string attributes:

```
>>> from tinydb import TagQuery
>>> Tags = TagQuery()
>>> db.search(Tags["country name"] == "United States of America")
```

6.1.4 Time Queries

To query based on time, the "right-hand side" of the `TimeQuery` should be a timezone-aware `datetime` object:

```
>>> from tinydb import TimeQuery
>>> from datetime import datetime, timezone
>>> Time = TimeQuery()
>>> db.search(Time > datetime(2000, 1, 1, tzinfo=timezone.utc))
```

To query for a range of timestamps, it is most-performant to combine two `TimeQuery` instances with the `&` operator (for more details on compound queries, see *Compound Queries and Query Modifiers* below):

```
>>> q1 = Time > datetime(1990, 1, 1, tzinfo=timezone.utc)
>>> q2 = Time < datetime(2020, 1, 1, tzinfo=timezone.utc)
>>> db.search(q1 & q2)
```

Note: Queries can be optimized for faster results. See *Tips for TinyFlux* for details on optimizing queries.

6.2 Advanced Simple Queries

Some queries require transformations or comparisons that go beyond the basic operators like ==, <, or >. To this end, TinyFlux supports the following queries:

.map(...) <–> Arbitrary Transform Functions for All Query Types

The `map()` method will transform the tag/field value, which will be compared against the right-hand side value from the query.

```
>>> # Get all points with a even value for 'number_of_pedals'.
>>> def mod2(value):
...     return value % 2
>>> Field = FieldQuery()
>>> db.search(Field.number_of_pedals.map(mod2) == 0)
```

or:

```
>>> # Get all points with a measurement starting with the letter "a".
>>> def get_first_letter(value):
...     return value[0]
>>> Measurement = MeasurementQuery()
>>> db.search(Measurement.map(get_first_letter) == "a")
```

Warning: Resist the urge to build your own time range query using the `.map()` query method. This will result in slow queries. Instead, use two `TimeQuery` instances combined with the `&` or `|` operator.

.test(...) <–> Arbitrary Test Functions for All Query Types

The `test()` method will transform and test the tag/field value for truthiness, with no right-hand side value necessary.

```
>>> # Get all points with a even value for 'number_of_pedals'.
>>> def is_even(value):
...     return value % 2 == 0
>>> Field = FieldQuery()
>>> db.search(Field.number_of_pedals.test(is_even))
```

or:

```
>>> # Get all points with a measurement starting with the letter "a".
>>> def starts_with_a(value):
...     return value.startswith("a")
>>> Measurement = MeasurementQuery()
>>> db.search(Measurement.test(starts_with_a))
```

.exists() <–> Existence of Tag Key or Field Key

This applies to `TagQuery` and `FieldQuery` only.

```
>>> Field, Tag = TagQuery(), FieldQuery()
>>> db.search(Tag.user_name.exists())
>>> db.search(Field.age.exists())
```

.matches(...) and .search(...) <–> Regular Expression Queries for Measurements and Tag Values

RegEx queries that apply to `MeasurementQuery` and `TagQuery` only.

```
>>> # Get all points with a user name containing "john", case-invariant.
>>> Tag = TagQuery()
>>> db.search(Tag.user_name.matches('.*john.*', flags=re.IGNORECASE))
```

6.3 Compound Queries and Query Modifiers

TinyFlux also allows supports compound queries through the use of logical operators. This is particularly useful for time queries when a time range is needed.

```
>>> from tinydb import TimeQuery
>>> from datetime import datetime, timezone
>>> Time = TimeQuery()
>>> q1 = Time > datetime(1990, 1, 1, tzinfo=timezone.utc)
>>> q2 = Time < datetime(2020, 1, 1, tzinfo=timezone.utc)
>>> db.search(q1 & q2)
```

The three supported logical operators are **logical-and**, **logical-or**, and **logical-not**.

6.3.1 Logical AND (”&”)

```
>>> # Logical AND:
>>> Time = TimeQuery()
>>> t1 = datetime(2010, 1, 1, tzinfo=timezone.utc)
>>> t2 = datetime(2020, 1, 1, tzinfo=timezone.utc)
>>> db.search((Time >= t1) & (Time < t2)) # Get all points in 2010's.
```

6.3.2 Logical OR (“|”)

```
>>> # Logical OR:
>>> db.search((Time < t1) | (Time > t2)) # Get all points outside 2010's.
```

6.3.3 Logical NOT (“~”)

```
>>> # Negate a query:
>>> Tag = TagQuery()
>>> db.search(~(Tag.city == 'LA')) # Get all points whose city is not "LA".
```

Hint: When using & or |, make sure you wrap your queries on both sides with parentheses or Python will confuse the syntax.

Also, when using negation (~) you’ll have to wrap the query you want to negate in parentheses.

While not aesthetically pleasing to the eye, the reason for these parenthesis is that Python’s binary operators (&, |, and ~) have a higher operator precedence than comparison operators (==, >, etc.). For this reason, syntax like `~User.name`

`== 'John'` is parsed by Python as `(~User.name) == 'John'` which will throw an exception. See the Python docs on operator precedence for details.

Note: You **cannot** use `and` as a substitue for `&`, `or` as a subsititue for `|`, or `not` as a substitute for `~`. The `and`, `or`, and `not` keywords are reserved in Python and cannot be overridden, as the `&`, `|`, and `~` operators have been for TinyFlux queries.

The query and search operations covered above:

Simple Queries	
`MeasurementQuery() == my_measurement`	Match any Point with the measurement `my_measurement`
`TimeQuery() < my_time_value`	Match any Point with a timestamp prior to `my_time_value`
`TagQuery().my_tag_key == my_tag_value`	Matches any Point with a tag key of `my_tag_key` mapping to a tag value of `my_tag_value`
`FieldQuery().my_field_key == my_field_value`	Matches any Point with a field key of `my_field_key` mapping to a field value of `my_field_value`
Advanced Simple Queries	
`FieldQuery().my_field.exists()`	Match any Point where a field called `my_field` exists
`FieldQuery().my_field.map()`	Transform and tag or field value for comparison to a right-hand side value.
`FieldQuery().my_field.test(func, *args)`	Matches any Point for which the function returns `True`
`FieldQuery().my_field.matches(regex)`	Match any Point with the whole field matching the regular expression
`FieldQuery().my_field.search(regex)`	Match any Point with a substring of the field matching the regular expression
Compound Queries and Query Modifiers	
`~(query)`	Match Points that don't match the query
`(query1) & (query2)`	Match Points that match both queries
`(query1) | (query2)`	Match Points that match at least one of the queries

CHAPTER

SEVEN

EXPLORING DATA

An understanding of how queries in TinyFlux work can be applied to several database operations.

7.1 Query-based Exploration

The primary method for query usage is through the `.search(query)`. Other useful search methods are below:

.contains(query) <–> Check if the database contains any Points matching a Query

This returns a simple boolean value and is the fastest search op.

```
>>> # Check if db contains any Points for Los Angeles after the start of 2022.
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo
>>> q1 = TagQuery().city == "Los Angeles"
>>> q2 = TimeQuery() >= datetime(2022, 1, 1, tzinfo = ZoneInfo("US/Pacific"))
>>> db.contains(q1 & q2)
```

.count(query) <–> Count the number of Points matching a Query

This returns an integer.

```
>>> # Count the number of Points for Los Angeles w/ a temp over 100 degrees.
>>> q1 = TagQuery().city == "Los Angeles"
>>> q2 = FieldQuery().temperature_f > 100.0
>>> db.count(q1 & q2)
```

.get(query) <–> Get the first Point in the database matching a Query

This returns a Point instance, or `None` if no Points were found.

```
>>> # Return the first Point in the db for LA w/ more than 1 inch of precipitaion.
>>> q1 = TagQuery().city == "Los Angeles"
>>> q3 = FieldQuery().preciptation > 1.0
>>> db.get(q1 & q3)
```

.search(query) <–> Get all the Points in the database matching a Query

This is the primary method for querying the database, and returns a list of Point instances, sorted by timestamp.

```
>>> # Get all Points in the DB for Los Angeles in 2022 in which the AQI was "hazardous".
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo
```

(continues on next page)

(continued from previous page)

```
>>> q1 = TagQuery().city == "Los Angeles"
>>> q2 = TimeQuery() >= datetime(2022, 1, 1, tzinfo = ZoneInfo("US/Pacific"))
>>> q3 = TimeQuery() < datetime(2023, 1, 1, tzinfo = ZoneInfo("US/Pacific"))
>>> q4 = FieldQuery().air_quality_index > 100 # hazardous is over 100
>>> db.search(q1 & q2 & q3 & q4)
```

.select(attributes, query) <–> Get attributes from Points in the database matching a Query

This returns a list of attributes from Points matching the Query. Similar to SQL "select".

```
>>> # Get the time, city, and air-quality index ("AQI") for all Points with an AQI over␣
↪100.
>>> q = FieldQuery().aqi > 100
>>> db.select("fields.aqi", q)
[132]
>>> db.select(("time", "city", "fields.aqi"), q)
[(datetime.datetime(2020, 9, 15, 8, 0, tzinfo=datetime.timezone.utc), "Los Angeles",␣
↪132)]
```

7.2 Attribute-based Exploration

The database can also be explored based on attributes, as opposed to queries.

.get_measurements() <–> Get all the measurements in the database

This returns an alphabetically-sorted list of measurements in the database.

```
>>> db.insert(Point(measurement="cities"))
>>> db.insert(Point(measurement="stock prices"))
>>> db.get_measurements()
>>> ["cities", "stock prices"]
```

.get_field_keys() <–> Get all the field keys in the database

This returns an alphabetically-sorted list of field keys in the database.

```
>>> db.insert(Point(fields={"temp_f": 50.2}))
>>> db.insert(Point(fields={"price": 2107.44}))
>>> db.get_field_keys()
["temp_f", "price"]
```

.get_field_values(field_key) <–> Get all the field values in the database

This returns all the values for a specified field_key, in order of insertion order in the database. This might be useful for determining a range of values a field could take.

```
>>> db.insert(Point(fields={"temp_f": 50.2}))
>>> db.insert(Point(fields={"price": 2107.44}))
>>> db.get_field_values("temp_f")
[50.2]
```

.get_tag_keys() <–> Get all the tag keys in the database

This returns an alphabetically-sorted list of tag keys in the database.

```
>>> db.insert(Point(tags={"city": "LA"}))
>>> db.insert(Point(tags={"company": "Amazon.com, Inc."}))
>>> db.get_tag_keys()
["city", "company"]
```

.get_tag_values([tag_key]) <–> Get all the tag values in the database

This returns all the values for a list of specified tag keys.

```
>>> db.insert(Point(tags={"city": "LA"}))
>>> db.insert(Point(tags={"company": "Amazon.com, Inc."}))
>>> db.get_tag_values()
{"city": ["Los Angeles"], "company": ["Amazon.com, Inc."]}
```

.get_timestamps() <–> Get all the timestamps in the database

This returns all the timestamps in the database by insertion order.

```
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo
>>> time_2022 = datetime(2022, 1, 1, tzinfo = ZoneInfo("US/Pacific"))
>>> time_1900 = datetime(1900, 1, 1, tzinfo = ZoneInfo("US/Pacific"))
>>> db.insert(Point(time=time_2022))
>>> db.insert(Point(time=time_1900))
>>> db.get_timestamps()
[datetime.datetime(2022, 1, 1, 8, 0, tzinfo=datetime.timezone.utc), datetime.
↪datetime(1900, 1, 1, 8, 0, tzinfo=datetime.timezone.utc)]
```

7.3 Full Dataset Exploration

Sometimes access to all the data is needed. There are two methods for doing so- one that brings in all the database items into memory, and one that provides a generator that iterates over items one at a time.

.all() <–> Get all of the points in the database

This returns all the points in the database by timestamp order. To retrieve by insertion order, pass `sorted=False` argument. This will bring all of the data into memory at once.

```
>>> db.all() # Points returned sorted by timestamp.
```

or

```
>>> db.all(sorted=False) # Points returned by insertion order.
```

iter(db) <–> Iterate over all the points in the database

This returns a generator over which point-by-point logic can be applied. This does not pull everything into memory.

```
>>> iter(db)
<generator object TinyFlux.__iter__ at 0x103e3d970>
>>> for point in db:
...     print(point)
Point(time=2022-01-01T08:00:00+00:00, measurement=_default)
Point(time=1900-01-01T08:00:00+00:00, measurement=_default)
```

The list of all the data exploration methods covered above:

Query-based Exploration	
`db.contains(query)`	Whether or not the database contains any points matching a query
`db.count(query)`	Count the number of points matching a query
`db.get(query)`	Get one point from the database matching a query
`db.search(query)`	Get all points from the database matching a query
`db.select(attrributes, query)`	Get attributes froms points matching a query
Attribute-based Exploration	
`db.get_measurements()`	Get the names of all measurements in the database
`db.get_timestmaps()`	Get all the timestamps from the database, by insertion order
`db.get_tag_keys()`	Get all tag keys from the database
`db.get_tag_values()`	Get all tag values from the database
`db.get_field_keys()`	Get all field keys from the database
`db.get_field_values()`	Get all field values from the database
Full Dataset Exploration	
`db.all()`	Get all points in the database
`iter(db)`	Return a generator for all points in the database

CHAPTER

EIGHT

UPDATING POINTS

Though updating time series data tends to occur much less frequently than with other types of data, TinyFlux nonetheless supports the update of data with two methods. Update by query with the `update()` method and update all with the `update_all()` method. Updates are provided to both methods through the keyword arguments of `measurement`, `time`, `tags`, or `fields`. The values for these arguments are either static values, or a `Callable` returning static values. See below for examples.

Note: If you are a developer, or are otherwise interested in how TinyFlux performs updates behind the scenes, see the *TinyFlux Design Principles* page.

To update individual points in TinyFlux, first provide a query to the `update()` method, followed by one or more attributes to update and their values as keyword arguments. For example, to update the measurement names in the database for all points whose measurement value is "cities" to "US Metros", use a static value to the `measurement` keyword argument:

```
>>> Measurement = MeasurementQuery()
>>> db.update(Measurement == "cities", measurement="US Metros")
```

To update all timestamps for the measurement "US Metros" to be shifted backwards in time by one year, use a callable as the `time` keyword argument instead of a static value:

```
>>> from datetime import timedelta
>>> Measurement = MeasurementQuery()
>>> db.update(Measurement == "US Metros", time=lambda x: x - timedelta(days=365))
```

To change all instances of "CA" to "California" in a point's tag set for the "US Metros" measurement:

```
>>> Measurement = MeasurementQuery()
>>> def california_updater(tags):
...     if "state" in tags and tags["state"] == "CA":
...         return {**tags, "state": "California"}
...     else:
...         return tags
>>> db.update(Measurement == "US Metros", tags=california_updater)
```

Field updates occur much the same way as tags. To update all items in the database, use `update_all()`. For example, to convert all temperatures from Fahrenheit to Celcius if the field `temp` exists:

```
>>> def fahrenheit_to_celcius(fields):
...     if "temp" in fields:
...         temp_f = fields["temp"]
```

(continues on next page)

(continued from previous page)

```
...         temp_c =  (temp_f - 32.0) * (5/9)
...         return {**fields, "temp": temp_c}
...     else:
...         return fields
>>> db.update_all(fields=fahrenheit_to_celcius)
```

Warning: Like all other operations in TinyFlux, you cannot roll back the actions of `update()` or `update_all()`. There is no confirmation step, no access-control mechanism that prevents non-admins from performing this action, nor are there automatic snapshots stored anywhere. If you need these kinds of features, TinyFlux is not for you.

to recap, these are the two methods supporting the updating of data.

Methods	
`db.update(query, ...)`	Update any point matching the input query.
`db.update_all(...)`	Update all points.

CHAPTER

NINE

REMOVING POINTS

TinyFlux supports the removal of data with two methods. To remove by query, the `remove()` method is provided, and to remove all, use the `remove_all()` method. See below for examples.

Note: If you are a developer, or are otherwise interested in how TinyFlux performs deletes behind the scenes, see the *TinyFlux Design Principles* page.

The following will remove all points with the measurement value of "US Metros":

```
>>> Measurement = MeasurementQuery()
>>> db.remove(Measurement == "US Metros")
```

The following is an example of a manual time-based eviction.

```
>>> from datetime import datetime, timedelta, timezone
>>> Time = TimeQuery()
>>> t = datetime.now(timezone.utc) - timedelta(days=7)
>>> # Remove all points older that seven days.
>>> db.remove(Time < t)
```

To remove everything in the database , invoke `remove_all()`:

```
>>> db.remove_all()
```

Warning: Like all other operations in TinyFlux, you cannot roll back the actions of `remove()` or `remove_all()`. There is no confirmation step, no access-control mechanism that prevents non-admins from performing this action, nor are there automatic snapshots stored anywhere. If you need these kinds of features, TinyFlux is not for you.

to recap, these are the two methods supporting the removal of data.

Methods	
`db.remove(query)`	Remove any point matching the input query.
`db.remove_all()`	Remove all points.

CHAPTER

TEN

WORKING WITH MEASUREMENTS

TinyFlux supports working with multiple measurements. A measurement is analagous to a "table" in traditional RDMS. By accessing TinyFlux through a measurement, the same database API is utilized, but with a filter for the passed measurement.

To access TinyFlux through a measurment, use `db.measurement(name)`:

```
>>> db = TinyFlux("my_db.csv")
>>> m = db.measurement("my_measurement")
>>> m.insert(Point(time=datetime(2022, 1, 1, tzinfo=timezone.utc), tags={"my_tag_key":
↪"my_tag_value"}))
>>> m.all()
[Point(time=2022-01-01T00:00:00+00:00, measurement=my_measurement, tags=my_tag_key:my_
↪tag_value)]
>>> for point in m:
>>>     print(point)
Point(time=2022-01-01T00:00:00+00:00, measurement=my_measurement, tags=my_tag_key:my_tag_
↪value)
```

Note: TinyFlux uses a measurement named `_default` as the default measurement.

To remove a measurement and all its points from a database, use:

```
>>> db.drop_measurement('my_measurement')
```

or

```
>>> m.remove_all()
```

To get a list with the names of all measurements in the database:

```
>>> db.get_measurements()
["my_measurement"]
```

CHAPTER

ELEVEN

TIMEZONES IN TINYFLUX

Timestamps going in and out of TinyFlux are of the Python `datetime` type. At the file storage layer, TinyFlux stores these timestamps as ISO formatted strings in UTC. For seasoned Python users, this will be a familiar practice, as they will already be using timezone aware datetime objects in all cases and used to converting to-and-from UTC.

Hint: If you aren't already using timezone-aware datetime objects, there is no better time to start than now.

Hint: TLDR: All timestamps should be input as timezone-aware datetime objects in the UTC timezone. If you need to keep information about the local timezone of the observation, store it as a tag. Skip to example 5 below for proper initialization.

To illustrate the way time is handled in TinyFlux, below are the five ways time could potentially be initialized by a user. The fifth and final example is "best practice":

1. `time` is not set by the user when the Point is initialized so its default value is `None`. AFTER it is inserted into TinyFlux, it is assigned a UTC timestamp corresponding to the time of insertion.

```
>>> from tinyflux import Point, TinyFlux
>>> db = TinyFlux("my_db.csv") # an empty db
>>> p = Point()
>>> p.time is None
True
>>> db.insert(p)
>>> p.time
datetime.datetime(2021, 10, 30, 13, 53, 552872, tzinfo=datetime.timezone.utc)
```

2. `time` is set with a value, but it is not a `datetime` object. TinyFlux raises an exception.

```
>>> Point(time="2022-01-01")
ValueError: Time must be datetime object.
```

3. `time` is set with a `datetime` object that is "timezone-naive". TinyFlux considers this time to be local to the timezone of the computer that is running TinyFlux and will convert this time to UTC using the `astimezone` attribute of the `datetime` module upon insertion. This will lead to confusion down the road if TinyFlux is running on a remote computer, or the user was annotating data for points corresponding to places in other timezones.

```
>>> from datetime import datetime
>>> # Example: Our computer is in Californa, but we are working with a dataset of
>>> #          air quality measurements for Beijing, China.
>>> # Here, AQI was measured at 1pm local time in Beijing on Aug 28, 2021.
```

(continues on next page)

(continued from previous page)

```
>>> p = Point(
...     time=datetime(2021, 8, 28, 13, 0), # 1pm, datetime-naive
...     tags={"city": "beijing"},
...     fields={"aqi": 118}
... )
>>> p.time
datetime.datetime(2021, 8, 28, 13, 0)
>>> # Insert the point into the database.
>>> db.insert(p)
>>> # The point is cast to UTC, assuming the time was local to California, not Beijing.
>>> p.time
datetime.datetime(2021, 8, 28, 20, 0, tzinfo=datetime.timezone.utc)
```

4. `time` is set with a `datetime` object that is timezone-aware but the timezone is not UTC- TinyFlux casts the time to UTC for internal storage and retrieval and the original timezone is lost (it is up to the user to cast the timezone again after retrieval).

```
>>> from tinyflux import Point, TinyFlux
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo
>>> db = TinyFlux("my_db.csv") # an empty db
>>> la_point = Point(
...     time=datetime(2000, 1, 1, tzinfo=ZoneInfo("US/Pacific")),
...     tags={"city": "Los Angeles"}
...     fields={"temp_f": 54.0}
... )
>>> ny_point = Point(
...     time=datetime(2000, 1, 1, tzinfo=ZoneInfo("US/Eastern")),
...     tags={"city": "New York City"}
...     fields={"temp_f": 15.0}
... )
>>> db.insert_multiple([la_point, ny_point])
>>> # Notice the time attributes no longer carry the timezone information:
>>> la_point.time
datetime.datetime(2000, 1, 1, 8, 0, tzinfo=datetime.timezone.utc)
>>> ny_point.time
datetime.datetime(2000, 1, 1, 5, 0, tzinfo=datetime.timezone.utc)
```

Hint: If you need to keep the original, non-UTC timezone along with the dataset, consider adding a `tag` to your point indicating the timezone, for easier conversion after retrieval. TinyFlux will not assume nor attempt to store the timezone of your data for you.

5. `time` is set with a `datetime` object that is timezone-aware and the timezone is UTC. This is the easiest way to handle time. If needed, infomation about the timezone is stored in a tag.

```
>>> from datetime import datetime, timezone
>>> from tinyflux import TinyFlux, Point
>>> from zoneinfo import ZoneInfo
>>> # Time now is 10am in Los Angeles, which is 6pm UTC:
>>> t = datetime.now(timezone.utc)
>>> t
```

(continues on next page)

(continued from previous page)

```
datetime.datetime(2022, 11, 9, 18, 0, 0, tzinfo=datetime.timezone.utc)
>>> # Store the time in UTC, but keep the timezone as a tag for later use.
>>> p = Point(
...     time=t,
...     tags={"room": "bedroom", "timezone": "America/Los_Angeles"},
...     fields={"temp": 72.0}
... )
>>> # Time is still UTC:
>>> p.time
datetime.datetime(2022, 11, 9, 18, 0, 0, tzinfo=datetime.timezone.utc)
>>> # To cast back to local time in Los Angeles:
>>> la_timezone = ZoneInfo(p.tags["timezone"])
>>> p.time.astimezone(la_timezone)
datetime.datetime(2022, 11, 9, 10, 0, tzinfo=zoneinfo.ZoneInfo(key='America/Los_Angeles
↪'))
```

CHAPTER

TWELVE

TIPS FOR TINYFLUX

Below are some tips to get the most out of TinyFlux.

12.1 Optimizing Queries

Unlike TinyDB, TinyFlux never pulls in the entirety of its data into memory (unless the `.all()` method is called). This has the benefit of reducing the memory footprint of the database, but means that database operations are usually I/O bound. By using an index, TinyFlux is able to construct a matching set of items from the storage layer without actually reading any of those items. For database operations that return Points, TinyFlux iterates over the storage, collects the items that belong in the set, deserializes them, and finally returns them to the caller.

This utlimately means that the smaller the set of matches, the less I/O TinyFlux must perform.

Hint: Queries that return smaller sets of matches perform best.

Warning: Resist the urge to build your own time range query using the `.map()` query method. This will result in slow queries. Instead, use two `TimeQuery` instances combined with the `&` or `|` operator.

12.2 Keeping The Index Intact

TinyFlux must build an index when it is initialized as it currently does not save the index upon closing. If the workflow for the session is read-only, then the index state will never be modified. If, however, a TinyFlux session consists of a mix of writes and reads, then the index will become invalid if at any time, a Point is inserted out of time order.

```
>>> from tinyflux import TinyFlux, Point
>>> from datetime import datetime, timedelta, timezone
>>> db = TinyFlux("my_db.csv")
>>> t = datetime.now(timezone.utc) # current time
>>> db.insert(Point(time=t))
>>> db.index.valid
True
>>> db.insert(Point(time=t - timedelta(hours=1))) # a Point out of time order
>>> db.index.valid
False
```

If `auto-index` is set to `True` (the default setting), then the next read will rebuild the index, which may just seem like a very slow query. For smaller datasets, reindexing is usually not noticeable.

Hint: If possible, Points should be inserted into TinyFlux in time-order.

12.3 Dealing with Growing Datasets

As concurrency is not a feature of TinyFlux, a growing database will incur increases in query and index-building times. When queries start to slow down a workflow, it might be time to "shard" or denormalize the data, or simply upgrade to a database server like InfluxDB.

For example, if a TinyFlux database currently holds Points for two separate measurements, consider making two separate databases, one for each measurement:

```
>>> from tinyflux import TinyFlux, Point, MeasurementQuery
>>> from datetime import datetime, timedelta, timezone
>>> db = TinyFlux("my_big_db.csv") # a growing db with two measurements
>>> db.count(MeasurementQuery() == "measurement_1")
70000
>>> db.count(MeasurementQuery() == "measurement_2")
85000
>>> new_db = TinyFlux("my_new_single_measurement_db.csv") # a new empty db
>>> for point in db:
>>>     if point.measurement == "measurement_2":
>>>         new_db.insert(point)
>>> db.remove(MeasurementQuery() == "measurement_2")
85000
>>> len(db)
70000
>>> len(new_db)
85000
```

Hint: When queries and indexes slow down a workflow, consider creating separate databases. Or, just migrate to InfluxDB.

CHAPTER

THIRTEEN

ELEMENTS OF DATA IN TINYFLUX

Data elements and terms in TinyFlux mostly mirror those of InfluxDB. The following is a list of TinyFlux terms and concepts. Click on a term, or read on below.

- *Point*
- *Timestamp*
- *Measurement*
- *Tag Set*
- *Tag Key*
- *Tag Value*
- *Field Set*
- *Field Key*
- *Field Value*

13.1 Point

The atomic data unit of TinyFlux. Consists of a *Measurement*, *Timestamp*, *Tag Set*, and a *Field Set*. In the primary disk CSV storage, all attributes are serialized to unicode using the system default encoding.

In Python:

```
>>> from tinyflux import Point
>>> from datetime import datetime, timezone
>>> p = Point(
...     time=datetime.now(timezone.utc),
...     measurement="thermostat home",
...     tags={
...         "location": "bedroom",
...         "scale": "fahrenheit",
...     },
...     fields={
...         "temp": "70.0",
...     }
... )
```

On disk:

```
2022-05-13T23:19:46.573233,thermostat home,_tag_location,bedroom,_tag_scale,fahrenheit,_
↪field_temp,70.0
```

13.2 Timestamp

The time associated with a *Point*. As an attribute of a *Point*, it is a Python datetime object. Regardless of its state, when it is inserted into a TinyFlux database, it will become a timezone aware object cast to the UTC timezone.

On disk, it is serialized as a ISO 8601 formatted string and occupies the first column of the default CSV storage class.

In Python:

```
>>> Point()
```

On disk:

```
2022-05-13T23:19:46.573233,_default
```

For details on time's relationship with TinyFlux, see *Timezones in TinyFlux*.

13.3 Measurement

A measurement is a collection of *Points*, much like a table in a relational database. It is a string in memory and on disk. TinyFlux provides a convenient method for interacting with the *Points* through the `db.measurement(...)` method.

In Python:

```
>>> Point(measurement="cities")
```

On disk:

```
2022-05-13T23:19:46.573233,cities
```

See *Working with Measurements* for more details.

13.4 Tag Set

A tag set (or "tags") is the collection of *tag keys* and *tag values* belonging to a *Point*. TinyFlux is schemaless, so any Point can contain zero, one, or more tag keys and associated tag values. Tag keys and tag values are both strings. Tag keys and their values map to Points with a hashmap in a TinyFlux index, providing for efficient retrieval. In a well-designed TinyFlux database, the number of distinct tag values should not be as numerous as the *field values*. On disk, tag sets occupy side-by-side columns- one for the tag key and one for the tag value.

In Python:

```
>>> Point(
...     tags={
...         "city": "LA",
...         "neighborhood": "Chinatown",
...         "food": "good",
```

(continues on next page)

(continued from previous page)

```
...     }
... )
```

On disk:

```
2022-05-13T23:19:46.573233,_default,_tag_city,LA,_tag_neighborhood,Chinatown,_tag_food,
↪good
```

13.5 Tag Key

A tag key is the identifier for a *Tag Value* in a *Tag Set*. On disk, a tag key is prepended with `_tag_`.

In the following, the tag key is `city`.

```
>>> tags = {"city": "Los Angeles"}
```

13.6 Tag Value

A tag value is the associated value for a tag key in a *Tag Set*. On disk, it occupies the column next to that of the its tag key.

In the following, the tag value is `Los Angeles`.

```
>>> tags = {"city": "Los Angeles"}
```

13.7 Field Set

A field set (or "fields") is the collection of *field keys* and *field values* belonging to a *Point*. TinyFlux is schemaless, so any Point can contain zero, one, or more field keys and associated field values. Field keys are strings while field values are numeric (in Python, `float` or `int`). Field keys and their values **do not** map to Points in a TinyFlux index as it is assumed that the number of their distinct values is too numerous. On disk, field sets occupy side-by-side columns- one for the field key and one for the field value.

In Python:

```
>>> Point(
...     fields={
...         "num_restaurants": 12,
...         "num_boba_shops": 3,
...     }
... )
```

On disk:

```
2022-05-13T23:19:46.573233,_default,_field_num_restaurants,12,_field_num_boba_shops,3
```

13.8 Field Key

A field key is the identifier for a *Field Value* in a *Field Set*. On disk, a field key is prepended with `_field_`.

In the following, the field key is `num_restaurants`.

```
>>> fields = {"num_restaurants": 12}
```

13.9 Field Value

A field value is the associated value for a *Field Key* in a *Field Set*. On disk, it occupies the column next to that of the its field key.

In the following, the field value is `12`.

```
>>> fields = {"num_restaurants": 12}
```

CHAPTER

FOURTEEN

TINYFLUX DESIGN PRINCIPLES

InfluxDB implements optimal design principles for time series data. Some of these design principles have associated tradeoffs in performance. Design principles are discussed below.

- *Prioritize High-Speed Writes*
- *Minimize Memory Footprint*
- *Prioritize Searches for Time*
- *Schemaless design*
- *IDs and Duplicates*

14.1 Prioritize High-Speed Writes

Time series data is often write-heavy, and in cases when a time series database is used as a real-time data store, the frequency of writes can be quite high. TinyFlux has been designed to minimize any disruptions to writing to disk in a single thread in as fast a manner as possible. To accomplish this, TinyFlux utilizes a default CSV store which supports nearly instantaneous appends, regardless of underlying file size. TinyFlux will also invalidate its index if upon any insert, the timestamp for a Point preceeds that of the most-recent insert. TinyFlux will not attempt to rebuild its index upon invalidation during a write op.

14.2 Minimize Memory Footprint

While it would be great if databases could live in memory, this is not a reasonable design choice for everyday users. TinyFlux has been designed to never read the entire contents of its storage into memory unless explicitly asked to do so, and to balance the need for fast querying with a small memory footprint, TinyFlux builds an internal index. This index is generally about 80% smaller than the memory required to hold the entire dataset in memory, and still allows for query performance to equal or surpass that of keeping the database in memory. For removals and updates, TinyFlux still visits all items in storage, but evaluates each item one at a time and writes to temporary storage before finally replacing the original storage with the updated one. TinyFlux also does not rewrite data in time-ascending order, as is the case with InfluxDB, as this would require either the entire dataset to be read into memory, or a computationally expensive eternal merge sort to be executed on disk.

14.3 Prioritize Searches for Time

TinyFlux builds an index on time by keeping a sorted container data structure of timestamps in memory, and searches over the index quickly by parsing queries and invoking optimized search algorithms for sorted containers to retrive candidate Points quickly. This reduces potentially slow and exhaustive evaluations significantly.

14.4 Schemaless design

Even though row-based data stores like CSV are not thought of as "schemaless", TinyFlux nonetheless allows for datasets to have flexible schemas so that signals that change over time, or multiple signals from multiple sources, can all occupy space in the same datastore. This allows the user to focus less on database design and more on capturing and analyzing data.

14.5 IDs and Duplicates

TinyFlux does not keep IDs as it is assumed data points are unique by their combination of timestamp and tag set. To this end, TinyFlux also does not currently have a mechanism for checking for duplicates. Searches matching duplicate Points will return duplicates.

CHAPTER

FIFTEEN

TINYFLUX INTERNALS

15.1 Storage

TinyFlux ships with two types of storage:

1. A CSV store with is persistent to disk, and
2. A memory store which lasts only as long as the process in which it was declared.

To use the CSV store, pass a filepath during TinyFlux initialization.

```
>>> my_database = "db.csv"
>>> db = TinyDB(my_database)
```

To use the memory store:

```
>>> from tinydb.storages import MemoryStorage
>>> db = TinyDB(storage=MemoryStorage)
```

In nearly all cases, users should opt for the former as it persists the data on disk.

The CSV format is familiar to most, but at its heart it's just a row-based datastore that supports sequential iteration and append-only writes. Contrast this with JSON, which–while fast once loaded into memory–must be loaded entirely into memory and does not support appending.

The usage of CSV offers TinyFlux two distinct advantages for typical time-series workflows:

1. Appends do not require reading of data, and occur in a constant amount of time regardless of the size of the underlying database.
2. Sequential iteration allows for a full read of the data without having to simulateously keep the entirety of the data store in memory all at once. Logic can be performed on an individual row, and results kept or discarded as desired.

TinyFlux storage is also designed to be extensible.

In case direct access to the storage instance is desired, use the `storage` property of the TinyFlux instance.

```
>>> from tinyflux.storages import MemoryStorage
>>> db = TinyFlux(storage=MemoryStorage)
>>> my_data = db.storage.read()
```

For more disucssion on storage, see *TinyFlux Design Principles*.

15.2 Indexing in TinyFlux

By default, TinyFlux will build an internal index when the database is initialized, and again at any point when a read operation is performed after the index becomes invalid. As TinyFlux's primary storage format is a CSV that is read from disk sequentially, the index allows for efficient retrieval operations that greatly reduce function calls, query evaluations, and the need to deserialize and reserialize data.

Note: An index becomes invalid when points are inserted out-of-time-order. When the `auto-index` parameter of `TinyFlux` is set to `True`, the next read operation will rebuild the index.

Building an index is a non-trivial routine that occurs in the same process that TinyFlux is running in. For smaller amounts of data in a typical analytics workflow, building an index may not even be noticeable. As the database grows, the time needed to build or rebuild the index grows linearly. Automatically rebuilding of the index can be turned off by setting `auto_index` to `False` in the TinyFlux constructor:

```
>>> db = TinyFlux("my_database.csv", auto_index=False)
```

Setting this value to `False` will remove any indexing-building, but queries will slow down considerably.

A reindex can be manually triggered should the need arise:

```
>>> db.reindex()
```

Warning: There is usually only one reason to turn off auto-indexing and that is when you are initializing the database instance and need to **immediately** start inserting points, as might be the case in IOT data-capture applications. In all other cases, particularly when reads will make up the majority of your workflow, you should leave `auto-index` set to `True`.

At some level of data, the building of the index will noticeably slow down a workflow. For tips on how to address growing data, see *Tips for TinyFlux*.

CHAPTER

SIXTEEN

TINYFLUX API

See *Getting Started* to get TinyFlux up and running with writing and querying data.

Jump to an API section:

- *TinyFlux Database API*
- *Point API*
- *Queries API*
- *Measurement API*
- *Index API*
- *Storages API*
- *Utils API*

16.1 TinyFlux Database API

The main module of the TinyFlux package, containing the TinyFlux class.

class tinyflux.database.**TinyFlux**(**args*, *auto_index=True*, ***kwargs*)

Bases: object

The TinyFlux class containing the interface for the TinyFlux package.

A facade singleton for the TinyFlux program. Manages the lifecycles of Storage, Index, and Measurement instances. Handles Points and Queries.

TinyFlux will reindex data in memory by default. To turn off this feature, set the value of 'auto_index' to false in the constructor keyword arguments.

TinyFlux will use the CSV store by default. To use a different store, pass a derived Storage subclass to the 'storage' keyword argument of the constructor.

All other args and kwargs are passed to the Storage instance.

Data Storage Model: Data in TinyFlux is represented as Point objects. These are serialized and inserted into the TinyFlux storage layer in append-only fashion, providing the lowest-latency write op possible. This is of primary importance for time-series data which can often be written at a high- frequency. The schema of the storage layer is not rigid, allowing for variable metadata structures to be stored to the same data store.

Attributes: storage: A reference to the Storage instance. index: A reference to the Index instance.

Usage:

```
>>> from tinyflux import TinyFlux
>>> db = TinyFlux("my_tf_db.csv")
```

all(*sorted=True*)
Get all data in the storage layer as Points.

Args: sorted: Whether or not to return points sorted by time.

Returns: A list of Points.

Return type List[*Point*]

close()
Close the database.

This may be needed if the storage instance used for this database needs to perform cleanup operations like closing file handles.

To ensure this method is called, the tinyflux instance can be used as a context manager:

```
>>> with TinyFlux('data.csv') as db:
        db.insert(Point())
```

Upon leaving this context, the 'close' method will be called.

Return type None

contains(*query*, *measurement=None*)
Check whether the database contains a point matching a query.

Defines a function that iterates over storage items and submits it to the storage layer.

Args: query: A Query. measurement: An optional measurement to filter by.

Returns: True if point found, else False.

Return type bool

count(*query*, *measurement=None*)
Count the points matching a query in the database.

Args: query: a Query. measurement: An optional measurement to filter by.

Returns: A count of matching points in the measurement.

Return type int

default_measurement_name = '_default'

default_storage_class
alias of *tinyflux.storages.CSVStorage*

drop_measurement(*name*)
Drop a specific measurement from the database.

If 'auto-index' is True, a new index will be built.

Args: name: The name of the measurement.

Returns: The count of removed items.

Raises: OSError if storage cannot be written to.

Return type int

get(*query*, *measurement=None*)
Get exactly one point specified by a query from the database.

Returns None if the point doesn't exist.

Args: query: A Query. measurement: An optional measurement to filter by.

Returns: First found Point or None.

Return type Optional[*Point*]

get_field_keys(*measurement=None*)
Get all field keys in the database.

Args: measurement: Optional measurement to filter by.

Returns: List of field keys, sorted.

Return type List[str]

get_field_values(*field_key*, *measurement=None*)
Get field values in the database.

Args: field_key: Field key to get values for. measurement: Optional measurement to filter by.

Returns: List of field values.

Return type List[Union[int, float, None]]

get_measurements()
Get the names of all measurements in the database.

Returns: Names of all measurements in storage as a set.

Return type List[str]

get_tag_keys(*measurement=None*)
Get all tag keys in the database.

Args: measurement: Optional measurement to filter by.

Returns: List of field keys, sorted.

Return type List[str]

get_tag_values(*tag_keys=[]*, *measurement=None*)
Get all tag values in the database.

Args: tag_keys: Optional list of tag keys to get associated values for. measurement: Optional measurement to filter by.

Returns: Mapping of tag_keys to associated tag values as a sorted list.

Return type Dict[str, List[str]]

get_timestamps(*measurement=None*)
Get all timestamps in the database.

Returns timestamps in order of insertion in the database, as time-aware datetime objects with UTC timezone.

Args: measurement: Optional measurement to filter by.

Returns: List of timestamps by insertion order.

Return type List[datetime]

property index: *tinyflux.index.Index*
Get a reference to the index instance.

Return type *Index*

insert(*point*, *measurement=None*)
Insert a Point into the database.

Args: point: A Point object. measurement: An optional measurement to filter by.

Returns: 1 if success.

Raises: OSError if storage cannot be appendex to. TypeError if point is not a Point instance.

Return type int

insert_multiple(*points*, *measurement=None*)
Insert Points into the database.

Args: points: An iterable of Point objects. measurement: An optional measurement to insert Points into.

Returns: The count of inserted points.

Raises: OSError if storage cannot be appendex to. TypeError if point is not a Point instance.

Return type int

measurement(*name*, ***kwargs*)
Return a reference to a measurement in this database.

Chained methods will be handled by the Measurement class, and operate on the subset of Points belonging to the measurement.

A measurement does not need to exist in the storage layer for a Measurement object to be created.

Args: name: Name of the measurement

Returns: Reference to the measurement.

Return type *Measurement*

reindex()
Build a new in-memory index.

Raises: OSError if storage cannot be written to.

Return type None

remove(*query*, *measurement=None*)
Remove Points from this database by query.

This is irreversible.

Args: query: A query to remove Points by. measurement: An optional measurement to filter by.

Returns: The count of removed points.

Raises: OSError if storage cannot be written to.

Return type int

remove_all()
Remove all Points from this database.

This is irreversible.

Raises: OSError if storage cannot be written to.

Return type None

search(*query*, *measurement=None*, *sorted=True*)
Get all points specified by a query.

Args: query: A Query. measurement: An optional measurement to filter by. sorted: Whether or not to return the points sorted by time.

Returns: A list of found Points.

Return type List[*Point*]

select(*select_keys*, *query*, *measurement=None*)
Get specified attributes from Points specified by a query.

'select_keys' should be an iterable of attributres including 'time', 'measurement', and tag keys and tag values. Passing 'tags' or 'fields' in the 'select_keys' iterable will not retrieve all tag and/or field values. Tag and field keys must be specified individually.

Args: select_keys: A Point attribute or iterable of Point attributes. query: A Query. measurement: An optional measurement to filter by.

Returns: A list of Point attribute values.

Return type List[Union[Any, Tuple[Any, ...]]]

property storage: *tinyflux.storages.Storage*
Get a reference to the storage instance.

Return type *Storage*

update(*query*, *time=None*, *measurement=None*, *tags=None*, *fields=None*, *_measurement=None*)
Update all matching Points in the database with new attributes.

Args: query: A query as a condition. time: A datetime object or Callable returning one. measurement: A string or Callable returning one. tags: A mapping or Callable returning one. fields: A mapping or Callable returning one. _measurement: An optional Measurement to filter by.

Returns: A count of updated points.

Raises: OSError if storage cannot be written to.

Return type int

update_all(*time=None*, *measurement=None*, *tags=None*, *fields=None*)
Update all points in the database with new attributes.

Args: time: A datetime object or Callable returning one. measurement: A string or Callable returning one. tags: A mapping or Callable returning one. fields: A mapping or Callable returning one.

Returns: A count of updated points.

Raises: OSError if storage cannot be written to.

Return type int

tinyflux.database.**append_op**(*method*)
Decorate an append operation with assertion.

Ensures storage can be appended to before doing anything.

tinyflux.database.**read_op**(*method*)
Decorate a read operation with assertion.

Ensures storage can be read from before doing anything.

tinyflux.database.**temp_storage_op**(*method*)
Decorate a db operation that requires auxiliary storage.

Initializes temporary storage, invokes method, and cleans-up storage after op has run.

tinyflux.database.**write_op**(*method*)
Decorate a write operation with assertion.

Ensures storage can be written to before doing anything.

16.2 Point API

Defintion of the TinyFlux Point class.

A Point is the data type upon which TinyFlux manages. It contains the time data and metadata for an individual observation. Points are serialized and deserialized from Storage. SimpleQuerys act upon individual Points.

A Point is comprised of a timestamp, a measurement, fields, and tags.

Fields contains string/numeric key-values, while tags contain string/string key-values. This is enforced upon Point instantiation.

Usage:

```
>>> from tinyflux import Point
>>> p = Point(
        time=datetime.now(timezone.utc),
        measurement="my measurement",
        fields={"my field": 123.45},
        tags={"my tag key": "my tag value"}
    )
```

class tinyflux.point.**Point**(**args*, ***kwargs*)

Bases: object

Define the Point class.

This is the only data type that TinyFlux handles directly. It is composed of a timestamp, measurement, tag-set, and field-set.

Usage:

```
>>> p = Point(
        time=datetime.now(timezone.utc),
        measurement="my measurement",
        fields={"my field": 123.45},
        tags={"my tag key": "my tag value"}
    )
```

default_measurement_name = '_default'

property fields

Get fields.

property measurement

Get measurement.

property tags

Get tags.

property time

Get time.

tinyflux.point.**validate_fields**(*fields*)

Validate fields.

Args: fields: The object to validate.

Raises: ValueError: Exception if fields cannot be validated.

Return type None

tinyflux.point.**validate_tags**(*tags*)

Validate tags.

Args: tags: The object to validate.

Raises: ValueError: Exception if tags cannot be validated.

Return type None

16.3 Queries API

Defintion of TinyFlux Queries.

A query contains logic in the form of a test and it acts upon a single Point when it is eventually evaluated.

All Queries begin as subclass of BaseQuery, which is not itself callable. Logic for the query is handled by the Python data model of the BaseQuery class, resulting in the generation of a SimpleQuery, which is callable. SimpleQuery instances support logical AND, OR, and NOT operations, which result in the initialization of a new CompoundQuery object.

Each SimpleQuery instance contains attributes that constitute the "deconstuction" of a query into several key parts (e.g. the operator, the right-hand side) so that the other consumers of queries, includng an Index, may use them for their own purposes.

class tinyflux.queries.**BaseQuery**

Bases: object

A base class for the different TinyFlux query types.

A query type that explicity unifies the divergent interfaces of TimeQuery, MeasurementQuery, TagQuery, and FieldQuery.

A BaseQuery is not iteslf callable. When it is combined with test logic, it generates a SimpleQuery, which is callable without exception.

Usage:

```
>>> from tinyflux import TagQuery, Point
>>> p = Point(tags={"city": "LA"})
>>> q1 = TagQuery()
>>> isinstance(q1, tinyflux.queries.BaseQuery)
True
>>> q1(p)
RuntimeError: Empty query was evaluated.
>>> q2 = TagQuery().city == "LA"
>>> q2
SimpleQuery('tags', '==', ('city',), 'LA')
>>> q2(p)
True
```

is_hashable()

Return hash is not empty.

Return type bool

map(*func*)

Add a function to the query path.

Similar to __getattr__ but for arbitrary functions.

Args: func: The function to add.

Return type *BaseQuery*

matches(*regex*, *flags=0*)

Run a regex test against a value (whole string has to match).

```
>>> TagQuery().f1.matches(r'^\\w+$')
```

Args: regex: The regular expression to use for matching. flags: Regex flags to pass to re.match.

Return type *SimpleQuery*

noop()
Evaluate to True.

Useful for having a base value when composing queries dynamically.

Return type *SimpleQuery*

search(*regex*, *flags=0*)
Run a regex test against a value (only substring has to match).

```
>>> TagQuery().f1.search(r'^\\w+$')
```

Args: regex: The regular expression to use for matching. flags: Regex flags to pass to re.match.

Return type *SimpleQuery*

test(*func*, **args*)
Run a user-defined test function against a value.

```
>>> def test_func(val):
...     return val == 42
...
>>> FieldQuery()["my field"].test(test_func)
```

Warning: The test fuction provided needs to be deterministic (returning the same value when provided with the same arguments), otherwise this may mess up the query cache that `Table` implements.

Args: func: The function to call, passing the value as the first arg. args: Additional arguments to pass to the test function.

Return type *SimpleQuery*

class tinyflux.queries.**CompoundQuery**(*query1*, *query2*, *operator*, *hashval*)
Bases: `object`

A container class for simple and/or compound queries and an operator.

A CompoundQuery is generated by built-in __and__, __or__, and __not__ operations on a SimpleQuery.

Attributes: query1: A SimpleQuery or CompoundQuery instance. query2: A SimpleQuery or CompoundQuery instance. operator: The operator.

Usage:

```
>>> from tinyflux import FieldQuery, TagQuery
>>> time_q = FieldQuery().temp_f < 55.0
>>> tags_q = TagQuery().city == "Los Angeles"
>>> cold_LA_q = time_q & tags_q
>>> type(cold_LA_q)
<class 'tinyflux.queries.CompoundQuery'>
```

is_hashable()
Return the ability to hash this query.

Return type bool

class tinyflux.queries.**FieldQuery**

Bases: *tinyflux.queries.BaseQuery*

The base query for Point fields.

Generates a SimpleQuery that evaluates Point 'fields' attributes.

Usage:

```
>>> from tinyflux import FieldQuery
>>> my_field_q = FieldQuery().my_field == 10.0
```

exists()

Test at Point where a provided key exists.

Usage:

```
>>> FieldQuery().my_field.exists()
```

Return type *SimpleQuery*

matches(*regex*, *flags=0*)

Raise an exception for regex query.

Return type *SimpleQuery*

search(*regex*, *flags=0*)

Raise an exception for regex query.

Return type *SimpleQuery*

class tinyflux.queries.**MeasurementQuery**

Bases: *tinyflux.queries.BaseQuery*

The base query for Point measurement.

Generates a SimpleQuery that evaluates Point 'measurement' attributes.

Usage:

```
>>> from tinyflux import MeasurementQuery
>>> my_measurement_q = MeasurementQuery() == "my measurement"
```

class tinyflux.queries.**SimpleQuery**(*point_attr*, *operator*, *rhs*, *test*, *path_resolver*, *hashval*)

Bases: object

A single query instance.

This is the object on which the actual query operations are performed. The BaseQuery class acts like a query builder and generates SimpleQuery objects which will evaluate their query against a given point when called.

Query instances can be combined using logical OR and AND and inverted using logical NOT.

A SimpleQuery can be parsed using private attributes.

TODO: In order to be usable in a query cache, a query needs to have a stable hash value with the same query always returning the same hash. That way a query instance can be used as a key in a dictionary.

Usage:

```
>>> from tinyflux import TagQuery
>>> los_angeles_q = TagQuery().city == "Los Angeles"
>>> type(los_angeles_q)
<class 'tinyflux.queries.SimpleQuery'>
```

is_hashable()
Return the ability to hash this query.

Return type bool

property point_attr: str
Get the attribute of a Point object relevant for this query.

Return type str

class tinyflux.queries.**TagQuery**
Bases: *tinyflux.queries.BaseQuery*

The base query for Point tags.

Generates a SimpleQuery that evaluates Point 'tags' attributes.

Usage:

```
>>> from tinyflux import TagQuery
>>> my_tag_q = TagQuery().my_tag_key == "my tag value"
```

exists()
Test a Point where a provided key exists.

```
>>> TagQuery().my_tag.exists()
```

Return type *SimpleQuery*

class tinyflux.queries.**TimeQuery**
Bases: *tinyflux.queries.BaseQuery*

The base query for Point time.

Generates a SimpleQuery that evaluates Point 'measurement' attributes.

Usage:

```
>>> from datetime import datetime, timezone
>>> from tinyflux import TimeQuery
>>> my_time_q = TimeQuery() < datetime.now(timezone.utc)
```

matches(*regex*, *flags=0*)
Raise an exception for regex query.

Return type *SimpleQuery*

search(*regex*, *flags=0*)
Raise an exception for regex query.

Return type *SimpleQuery*

16.4 Measurement API

Defintion of TinyFlux measurement class.

The measurement class provides a convenient interface into a subset of data points with a common measurement name. A measurement is analogous to a table in a traditional RDBMS.

Usage:

```
>>> db = TinyFlux(storage=MemoryStorage)
>>> m = db.measurement("my_measurement")
```

class tinyflux.measurement.**Measurement**(*name*, *db*)

Bases: object

Define the Measurement class.

Measurement objects are created at runtime when the TinyFlux 'measurement' method is invoked.

Attributes: name: Name of the measurement. storage: Storage object for the measurement's parent TinyFlux db. index: Index object for the measurement's parent TinyFlux db.

all(*sorted=True*)

Get all points in this measurement.

Args: sorted: Whether or not to return points in sorted time order.

Returns: A list of points.

Return type List[*Point*]

contains(*query*)

Check whether the measurement contains a point matching a query.

Args: query: A SimpleQuery.

Returns: True if point found, else False.

Return type bool

count(*query*)

Count the points matching a query in this measurement.

Args: query: a SimpleQuery.

Returns: A count of matching points in the measurement.

Return type int

get(*query*)

Get exactly one point specified by a query from this measurement.

Returns None if the point doesn't exist.

Args: query: A SimpleQuery.

Returns: First found Point or None.

Return type Optional[*Point*]

get_field_keys()
Get all field keys for this measurement.

Returns: List of field keys, sorted.

Return type List[str]

get_field_values(*field_key*)
Get field values from this measurement for the specified key.

Args: field_key: The field key to get field values for.

Returns: List of field keys, sorted.

Return type List[Union[int, float, None]]

get_tag_keys()
Get all tag keys for this measurement.

Returns: List of tag keys, sorted.

Return type List[str]

get_tag_values(*tag_keys=[]*)
Get all tag values in the database.

Args: tag_keys: Optional list of tag keys to get associated values for.

Returns: Mapping of tag_keys to associated tag values as a sorted list.

Return type Dict[str, List[str]]

get_timestamps()
Get all timestamps in the database.

Returns timestamps in order of insertion in the database, as time-aware datetime objects with UTC time-zone.

Args: measurement: Optional measurement to filter by.

Returns: List of timestamps by insertion order.

Return type List[datetime]

property index: *tinyflux.index.Index*
Get the measurement storage instance.

Return type *Index*

insert(*point*)
Insert a Point into a measurement.

If the passed Point has a different measurement value, 'insert' will update the measurement value with that of this measurement.

Args: point: A Point object.

Returns: 1 if success.

Raises: TypeError if point is not a Point instance.

Return type int

insert_multiple(*points*)
Insert Points into this measurement.

If the passed Point has a different measurement value, 'insert' will update the measurement value with that of this measurement.

Args: points: An iterable of Point objects.

Returns: The count of inserted points.

Raises: TypeError if point is not a Point instance.

Return type int

property name: str
Get the measurement name.

Return type str

remove(*query*)
Remove Points from this measurement by query.

This is irreversible.

Returns: The count of removed points.

Return type int

remove_all()
Remove all Points from this measurement.

This is irreversible.

Returns: The count of removed points.

Return type int

search(*query*, *sorted=True*)
Get all points specified by a query from this measurement.

Order is not guaranteed. Returns empty list if no points are found.

Args: query: A SimpleQuery. sorted: Whether or not to return points sorted by timestamp.

Returns: A list of found Points.

Return type List[*Point*]

select(*keys*, *query*)
Get specified attributes from Points specified by a query.

'keys' should be an iterable of attributres including 'time', 'measurement', and tag keys and tag values. Passing 'tags' or 'fields' in the 'keys' iterable will not retrieve all tag and/or field values. Tag and field keys must be specified individually.

Args: keys: An iterable of Point attributes. query: A Query.

Returns: A list of tuples of Point attribute values.

Return type List[Tuple[Union[datetime, str, int, float, None]]]

property storage: *tinyflux.storages.Storage*
Get the measurement storage instance.

Return type *Storage*

update(*query*, *time=None*, *measurement=None*, *tags=None*, *fields=None*)
Update all matching Points in this measurement with new attributes.

Args: query: A query. time: A datetime object or Callable returning one. measurement: A string or Callable returning one. tags: A mapping or Callable returning one. fields: A mapping or Callable returning one.

Returns: A count of updated points.

Return type `int`

update_all(*time=None*, *measurement=None*, *tags=None*, *fields=None*)
Update all matching Points in this measurement with new attributes.

Args: query: A query. time: A datetime object or Callable returning one. measurement: A string or Callable returning one. tags: A mapping or Callable returning one. fields: A mapping or Callable returning one.

Returns: A count of updated points.

Return type `int`

16.5 Index API

Defintion of the TinyFlux Index.

Class descriptions for Index and IndexResult. An Index acts like a singleton, and is initialized at creation time with the TinyFlux instance. It provides efficient in-memory data structures and getters for TinyFlux operations. An Index instance is not a part of the TinyFlux interface.

An IndexResult returns the indicies of revelant TinyFlux queries for further handling, usually as an input to a storage retrieval.

class `tinyflux.index.`**Index**(*valid=True*)
Bases: `object`

An in-memory index for the storage instance.

Provides efficient data structures and searches for TinyFlux data. An Index instance is created and its lifetime is handled by a TinyFlux instance.

Attributes: empty: Index contains no items (used in testing). valid: Index represents current state of TinyFlux.

build(*points*)
Build the index from scratch.

Args: points: The collection of points to build the Index from.

Usage:

```
>>> i = Index().build([Point()])
```

Return type None

property empty: bool
Return True if index is empty.

Return type bool

get_field_keys(*measurement=None*)
Get field keys from this index, optionally filtered by measurement.

Args: measurement: Optional measurement to filter by.

Returns: Set of field keys.

Return type Set[str]

get_field_values(*field_key*, *measurement=None*)
Get field values from this index, optionally filter by measurement.

Args: field_key: Field key to get field values for. measurement: Optional measurement to filter by.

Returns: List of field values.

Return type List[Union[int, float, None]]

get_measurements()
Get the names of all measurements in the Index.

Returns: Unique names of measurements as a set.

Usage:

```
>>> n = Index().build([Point()]).get_measurements()
```

Return type Set[str]

get_tag_keys(*measurement=None*)
Get tag keys from this index, optionally filtered by measurement.

Args: measurement: Optional measurement to filter by.

Returns: Set of field keys.

Return type Set[str]

get_tag_values(*tag_keys=[]*, *measurement=None*)
Get all tag values from the index.

Args: tag_keys: Optional list of tag keys to get associated values for. measurement: Optional measurement to filter by.

Returns: Mapping of tag_keys to associated tag values as a set.

Return type Dict[str, Set[str]]

get_timestamps(*measurement=None*)
Get timestamps from the index.

Args: measurement: Optional measurement to filter by.

Returns: List of timestamps.

Return type List[float]

insert(*points=[]*)
Update index with new points.

Accepts new points to add to an Index. Points are assumed to be passed to this method in non-descending time order.

Args: points: List of tinyflux.Point instances.

Usage:

```
>>> Index().insert([Point()])
```

Return type None

invalidate()
Invalidate an Index.

This method is invoked when the Index no longer represents the current state of TinyFlux and its Storage instance.

Usage:

```
>>> i = Index()
>>> i.invalidate()
```

Return type None

property lateset_time: datetime.datetime
Return the lastest time in the index.

Return type datetime

remove(*r_items*)
Remove items from the index.

Return type None

search(*query*)
Handle a TinyFlux query.

Parses the query, generates a new IndexResult, and returns it.

Args: query: A tinyflux.queries.Query.

Returns: An IndexResult instance.

Usage:

```
>>> i = Index().build([Point()])
>>> q = TimeQuery() < datetime.now(timezone.utc)
>>> r = i.search(q)
```

Return type *IndexResult*

update(*u_items*)
Update the index.

Args: u_items: A mapping of old indices to update indices.

Return type None

property valid: bool
Return an empty index.

Return type bool

class tinyflux.index.**IndexResult**(*items*, *index_count*)
Bases: object

Returns indicies of TinyFlux queries that are handled by an Index.

IndexResults instances are generated by an Index.

Arritributes: items: A set of indicies as ints.

Usage:

```
>>> IndexResult(items=set(), index_count=0)
```

property items: Set[int]
Return query result items.

Return type Set[int]

16.6 Storages API

Defintion of TinyFlux storages classes.

Storage defines an abstract base case using the built-in ABC of python. This class defines the requires abstract methods of read, write, and append, as well as getters and setters for attributes required to reindex the data.

A storage object will manage data with a file handle, or in memory.

A storage class is provided to the TinyFlux facade as an initial argument. The TinyFlux instance will manage the lifecycle of the storage instance.

Usage:

```
>>> my_mem_db = TinyFlux(storage=MemoryStorage)
>>> my_csv_db = TinyFlux('path/to/my.csv', storage=CSVStorage)
```

class tinyflux.storages.**CSVStorage**(*path*, *create_dirs=False*, *encoding=None*, *access_mode='r+'*, *flush_on_insert=True*, *newline=''*, ***kwargs*)
Bases: *tinyflux.storages.Storage*

Define the default storage instance for TinyFlux, a CSV store.

CSV provides append-only writes, which is efficient for high-frequency writes, common to time-series datasets.

Usage:

```
>>> from tinyflux import CSVStorage
>>> db = TinyFlux("my_csv_store.csv", storage=CSVStorage)
```

append(*items*, *temporary=False*)
Append points to the CSV store.

Args: items: A list of objects. temporary: Whether or not to append to temporary storage.

Return type None

property can_append: bool
Return whether or not appends can occur.

Return type bool

property can_read: bool
Return whether or not reads can occur.

Return type bool

property can_write: bool
Return whether or not writes can occur.

Return type bool

close()
Clean up data store.

Closes the file object.

Return type None

read()
Read all items from the storage into memory.

Returns: A list of Point objects.

Return type List[*Point*]

reset()
Reset the storage instance.

Removes all data.

Return type None

class tinyflux.storages.**MemoryStorage**
Bases: *tinyflux.storages.Storage*

Define the in-memory storage instance for TinyFlux.

Memory is cleaned up along with the parent process.

Attributes: _initially_empty: No data in the storage instance. _memory: List of Points. _temp_memory: List of Points.

Usage:

```
>>> from tinyflux import MemoryStorage
>>> db = TinyFlux(storage=MemoryStorage)
```

append(*items*, *temporary=False*)
Append points to the memory.

Args: points: A list of Point objects. temporary: Whether or not to append to temporary storage.

Return type None

read()
Read data from the store.

Returns: A list of Point objects.

Return type List[*Point*]

reset()
Reset the storage instance.

Removes all data.

Return type None

class tinyflux.storages.**Storage**
Bases: abc.ABC

The abstract base class for all storage types for TinyFlux.

Defines an extensible, static interface with required read/write ops and index-related getter/setters.

Custom storage classes should inheret like so:

```
>>> from tinyflux import Storage
>>> class MyStorageClass(Storage):
        ...
```

abstract append(*points*, *temporary=False*)
Append points to the store.

Args: points: A list of Point objets. temporary: Whether or not to append to temporary storage.

Return type None

property can_append: bool
Can append to DB.

Return type bool

property can_read: bool
Can read the DB.

Return type bool

property can_write: bool
Can write to DB.

Return type bool

close()
Perform clean up ops.

Return type None

abstract read()
Read from the store.

Re-ordering the data after a read provides TinyFlux with the ability to build an index.

Args: reindex_on_read: Reorder the store after data is read.

Returns: A list of Points.

Return type List[*Point*]

abstract reset()
Reset the storage instance.

Removes all data.

Return type None

tinyflux.storages.**create_file**(*path*, *create_dirs*)
Create a file if it doesn't exist yet.

Args: path: The file to create. create_dirs: Whether to create all missing parent directories.

Return type None

16.7 Utils API

Defintion of TinyFlux utils.

class tinyflux.utils.**FrozenDict**
Bases: dict

An immutable dictionary.

This is used to generate stable hashes for queries that contain dicts. Usually, Python dicts are not hashable because they are mutable. This class removes the mutability and implements the __hash__ method.

From TinyDB.

clear(**args*, ***kws*)
Raise a TypeError for a given dict method.

pop(*k*, *d=None*)
Raise TypeError for pop.

popitem(**args*, ***kws*)
Raise a TypeError for a given dict method.

update(*e=None*, ***f*)
Raise TypeError for update.

tinyflux.utils.**find_eq**(*sorted_list*, *x*)
Locate the leftmost value exactly equal to x.

Args: sorted_list: The list to search. x: The element to search.

Returns: The index of the found element or None.

Return type Optional[int]

tinyflux.utils.**find_ge**(*sorted_list*, *x*)
Find leftmost item greater than or equal to x.

Args: sorted_list: The list to search. x: The element to search.

Returns: The index of the found element or None.

Return type Optional[int]

tinyflux.utils.**find_gt**(*sorted_list*, *x*)
Find leftmost value greater than x.

Args: sorted_list: The list to search. x: The element to search.

Returns: The index of the found element or None.

Return type Optional[int]

tinyflux.utils.**find_le**(*sorted_list*, *x*)
Find rightmost value less than or equal to x.

Args: sorted_list: The list to search. x: The element to search.

Returns: The index of the found element or None.

Return type Optional[int]

tinyflux.utils.**find_lt**(*sorted_list*, *x*)
Find rightmost value less than x.

Args: sorted_list: The list to search. x: The element to search.

Returns: The index of the found element or None.

Return type Optional[int]

tinyflux.utils.**freeze**(*obj*)
Freeze an object by making it immutable and thus hashable.

Args: obj: Any python object.

Returns: The object in a hashable form.

Return type object

CHAPTER

SEVENTEEN

PHILOSOPHY

Like TinyDB, TinyFlux aims to be simple and fun to use.

Like InfluxDB, TinyFlux places time before all else.

Simplicity, enjoyment, and time- these are the three guiding principles of TinyFlux, both in its usage and in its development.

Finally, when in doubt, over-document your code.

CHAPTER

EIGHTEEN

GUIDELINES

New ideas, improvements, bugfixes, and new developer tools are always welcome. Follow these guidelines before getting started:

1. Make sure to read *Getting Started* and *Tooling and Conventions*.
2. Check GitHub for existing open issues, or open a new issue to begin a discussion.
3. To get started on a pull request, fork the repository on GitHub, create a new branch, and make updates.
4. Write unit tests, ensure the code is 100% covered, update documentation where necessary, and format and style the code correctly.
5. Send a pull request.

CHAPTER

NINETEEN

TOOLING AND CONVENTIONS

TinyFlux should be developed locally with the latest stable version of Python on any platform (3.10 as of this writing).

19.1 Versioning

TinyFlux follows semantic versioning guidelines for releases.

19.2 Workflow

TinyFlux development follows the branch-based workflow known as "GitHub flow".

19.3 Continuous Integration and Deployment

TinyFlux uses GitHub Actions for its CI/CD workflow.

19.4 Coding Conventions

TinyFlux conforms to PEP 8 for style, and Google Python Style Guide for docstrings. TinyFlux uses common developer tools to check and enforce this. These checks should be performed locally before pushing to GitHub, as they will eventually be enforced with GitHub Actions (see `.github/workflows` in the TinyFlux GitHub repository for details).

19.4.1 Formatting

TinyFlux uses standard configuration black for code formatting, with an enforced line-length of 79 characters.

After installing the project requirements:

```
/tinyflux $ black .
```

19.4.2 Style

TinyFlux uses standard configuration flake8 for style enforcement, with an enforced line-length of 79 characters.

After installing the project requirements:

```
/tinyflux $ flake8 .
```

19.4.3 Typing

TinyFlux uses standard configuration mypy for static type checking.

After installing the project requirements:

```
/tinyflux $ mypy .
```

19.4.4 Documentation

TinyFlux hosts documentation on Read The Docs.

TinyFlux uses Sphinx for documentation generation, with a customized Read the Docs Sphinx Theme, enabled for "Google-style" docstrings.

After installing the project requirements:

```
/tinyflux $ cd docs
/docs $ make html
/docs $ open build/html/index.html
```

Documentation is deployed to ReadTheDocs through third-party integration with GitHub. Commits to the `master` branch trigger builds and deployment with RTD.

19.5 Testing

TinyFlux aims for 100% code coverage through unit testing.

19.5.1 Test Framework

TinyFlux uses pytest as its testing framework.

After installing the project requirements:

```
/tinyflux $ pytest
```

19.5.2 Coverage

TinyFlux uses Coverage.py for measuring code coverage.

```
/tinyflux $ coverage run -m pytest
/tinyflux $ coverage report -m
```

CHAPTER

TWENTY

CHANGELOG

20.1 v0.2.1 (2022-11-22)

- Fix bug that caused values of 0.0 to be serialized as None/null rather than “0.0”.

20.2 v0.2.0 (2022-11-09)

- Test and verification on Python 3.11 and Windows platforms
- Disable universal newlines translation on CSV Storage instances

20.3 v0.1.0 (2022-05-16)

- Initial release

PYTHON MODULE INDEX

t

tinyflux.database, 47
tinyflux.index, 61
tinyflux.measurement, 58
tinyflux.point, 52
tinyflux.queries, 54
tinyflux.storages, 64
tinyflux.utils, 67

INDEX

A

all() (*tinyflux.database.TinyFlux method*), 48
all() (*tinyflux.measurement.Measurement method*), 58
append() (*tinyflux.storages.CSVStorage method*), 64
append() (*tinyflux.storages.MemoryStorage method*), 65
append() (*tinyflux.storages.Storage method*), 66
append_op() (*in module tinyflux.database*), 52

B

BaseQuery (*class in tinyflux.queries*), 54
build() (*tinyflux.index.Index method*), 61

C

can_append (*tinyflux.storages.CSVStorage property*), 65
can_append (*tinyflux.storages.Storage property*), 66
can_read (*tinyflux.storages.CSVStorage property*), 65
can_read (*tinyflux.storages.Storage property*), 66
can_write (*tinyflux.storages.CSVStorage property*), 65
can_write (*tinyflux.storages.Storage property*), 66
clear() (*tinyflux.utils.FrozenDict method*), 67
close() (*tinyflux.database.TinyFlux method*), 48
close() (*tinyflux.storages.CSVStorage method*), 65
close() (*tinyflux.storages.Storage method*), 66
CompoundQuery (*class in tinyflux.queries*), 55
contains() (*tinyflux.database.TinyFlux method*), 48
contains() (*tinyflux.measurement.Measurement method*), 58
count() (*tinyflux.database.TinyFlux method*), 48
count() (*tinyflux.measurement.Measurement method*), 58
create_file() (*in module tinyflux.storages*), 67
CSVStorage (*class in tinyflux.storages*), 64

D

default_measurement_name (*tinyflux.database.TinyFlux attribute*), 48
default_measurement_name (*tinyflux.point.Point attribute*), 53
default_storage_class (*tinyflux.database.TinyFlux attribute*), 48
drop_measurement() (*tinyflux.database.TinyFlux method*), 48

E

empty (*tinyflux.index.Index property*), 62
exists() (*tinyflux.queries.FieldQuery method*), 56
exists() (*tinyflux.queries.TagQuery method*), 57

F

FieldQuery (*class in tinyflux.queries*), 56
fields (*tinyflux.point.Point property*), 53
find_eq() (*in module tinyflux.utils*), 67
find_ge() (*in module tinyflux.utils*), 67
find_gt() (*in module tinyflux.utils*), 67
find_le() (*in module tinyflux.utils*), 68
find_lt() (*in module tinyflux.utils*), 68
freeze() (*in module tinyflux.utils*), 68
FrozenDict (*class in tinyflux.utils*), 67

G

get() (*tinyflux.database.TinyFlux method*), 49
get() (*tinyflux.measurement.Measurement method*), 58
get_field_keys() (*tinyflux.database.TinyFlux method*), 49
get_field_keys() (*tinyflux.index.Index method*), 62
get_field_keys() (*tinyflux.measurement.Measurement method*), 58
get_field_values() (*tinyflux.database.TinyFlux method*), 49
get_field_values() (*tinyflux.index.Index method*), 62
get_field_values() (*tinyflux.measurement.Measurement method*), 59
get_measurements() (*tinyflux.database.TinyFlux method*), 49
get_measurements() (*tinyflux.index.Index method*), 62
get_tag_keys() (*tinyflux.database.TinyFlux method*), 49
get_tag_keys() (*tinyflux.index.Index method*), 62
get_tag_keys() (*tinyflux.measurement.Measurement method*), 59
get_tag_values() (*tinyflux.database.TinyFlux method*), 49
get_tag_values() (*tinyflux.index.Index method*), 62
get_tag_values() (*tinyflux.measurement.Measurement method*), 59

get_timestamps() (*tinyflux.database.TinyFlux method*), 49
get_timestamps() (*tinyflux.index.Index method*), 62
get_timestamps() (*tinyflux.measurement.Measurement method*), 59

I

Index (*class in tinyflux.index*), 61
index (*tinyflux.database.TinyFlux property*), 50
index (*tinyflux.measurement.Measurement property*), 59
IndexResult (*class in tinyflux.index*), 64
insert() (*tinyflux.database.TinyFlux method*), 50
insert() (*tinyflux.index.Index method*), 63
insert() (*tinyflux.measurement.Measurement method*), 59
insert_multiple() (*tinyflux.database.TinyFlux method*), 50
insert_multiple() (*tinyflux.measurement.Measurement method*), 60
invalidate() (*tinyflux.index.Index method*), 63
is_hashable() (*tinyflux.queries.BaseQuery method*), 54
is_hashable() (*tinyflux.queries.CompoundQuery method*), 55
is_hashable() (*tinyflux.queries.SimpleQuery method*), 57
items (*tinyflux.index.IndexResult property*), 64

L

lateset_time (*tinyflux.index.Index property*), 63

M

map() (*tinyflux.queries.BaseQuery method*), 54
matches() (*tinyflux.queries.BaseQuery method*), 54
matches() (*tinyflux.queries.FieldQuery method*), 56
matches() (*tinyflux.queries.TimeQuery method*), 57
Measurement (*class in tinyflux.measurement*), 58
measurement (*tinyflux.point.Point property*), 53
measurement() (*tinyflux.database.TinyFlux method*), 50
MeasurementQuery (*class in tinyflux.queries*), 56
MemoryStorage (*class in tinyflux.storages*), 65
module
    tinyflux.database, 47
    tinyflux.index, 61
    tinyflux.measurement, 58
    tinyflux.point, 52
    tinyflux.queries, 54
    tinyflux.storages, 64
    tinyflux.utils, 67

N

name (*tinyflux.measurement.Measurement property*), 60
noop() (*tinyflux.queries.BaseQuery method*), 55

P

Point (*class in tinyflux.point*), 52
point_attr (*tinyflux.queries.SimpleQuery property*), 57
pop() (*tinyflux.utils.FrozenDict method*), 67
popitem() (*tinyflux.utils.FrozenDict method*), 67

R

read() (*tinyflux.storages.CSVStorage method*), 65
read() (*tinyflux.storages.MemoryStorage method*), 65
read() (*tinyflux.storages.Storage method*), 66
read_op() (*in module tinyflux.database*), 52
reindex() (*tinyflux.database.TinyFlux method*), 50
remove() (*tinyflux.database.TinyFlux method*), 50
remove() (*tinyflux.index.Index method*), 63
remove() (*tinyflux.measurement.Measurement method*), 60
remove_all() (*tinyflux.database.TinyFlux method*), 51
remove_all() (*tinyflux.measurement.Measurement method*), 60
reset() (*tinyflux.storages.CSVStorage method*), 65
reset() (*tinyflux.storages.MemoryStorage method*), 66
reset() (*tinyflux.storages.Storage method*), 66

S

search() (*tinyflux.database.TinyFlux method*), 51
search() (*tinyflux.index.Index method*), 63
search() (*tinyflux.measurement.Measurement method*), 60
search() (*tinyflux.queries.BaseQuery method*), 55
search() (*tinyflux.queries.FieldQuery method*), 56
search() (*tinyflux.queries.TimeQuery method*), 57
select() (*tinyflux.database.TinyFlux method*), 51
select() (*tinyflux.measurement.Measurement method*), 60
SimpleQuery (*class in tinyflux.queries*), 56
Storage (*class in tinyflux.storages*), 66
storage (*tinyflux.database.TinyFlux property*), 51
storage (*tinyflux.measurement.Measurement property*), 61

T

TagQuery (*class in tinyflux.queries*), 57
tags (*tinyflux.point.Point property*), 53
temp_storage_op() (*in module tinyflux.database*), 52
test() (*tinyflux.queries.BaseQuery method*), 55
time (*tinyflux.point.Point property*), 53
TimeQuery (*class in tinyflux.queries*), 57
TinyFlux (*class in tinyflux.database*), 47
tinyflux.database
    module, 47
tinyflux.index
    module, 61
tinyflux.measurement

module, 58
tinyflux.point
module, 52
tinyflux.queries
module, 54
tinyflux.storages
module, 64
tinyflux.utils
module, 67

U

update() (*tinyflux.database.TinyFlux method*), 51
update() (*tinyflux.index.Index method*), 63
update() (*tinyflux.measurement.Measurement method*), 61
update() (*tinyflux.utils.FrozenDict method*), 67
update_all() (*tinyflux.database.TinyFlux method*), 52
update_all() (*tinyflux.measurement.Measurement method*), 61

V

valid (*tinyflux.index.Index property*), 64
validate_fields() (*in module tinyflux.point*), 53
validate_tags() (*in module tinyflux.point*), 53

W

write_op() (*in module tinyflux.database*), 52