TinyDB Documentation

Release 1.3.0

Markus Siemens

July 25, 2014

Contents

```
>>> from tinydb import TinyDB, where
>>> db = TinyDB('/path/to/db.json')
>>> db.insert({'int': 1, 'char': 'a'})
>>> db.search(where('int') == 1)
[{'int': 1, 'char': 'a'}]
```

Welcome to TinyDB, your tiny, document oriented database optimized for your happiness :) If you're new here, consider reading the *introduction*, otherwise feel free to choose one of:

CHAPTER 1

Is TinyDB the right choice?

1.1 Why using TinyDB?

TinyDB is:

- **tiny**: The current source code has 800 lines of code (+ 500 lines tests) what makes about 100 KB. For comparison: Buzhug has about 2000 lines of code (w/o tests), CodernityDB has about 8000 lines of code (w/o tests).
- **document oriented**: Like MongoDB, you can store any document (represented as `dict`) in TinyDB.
- **optimized for your happiness**: TinyDB is designed to be simple and fun to use. It's not bloated and has a simple and clean API.
- **written in pure Python**: TinyDB neither needs an external server (as e.g. PyMongo) nor any packages from PyPI. Just run `pip install tinydb` and you're ready to go.
- **easily extensible**: You can easily extend TinyDB by writing new storages or modify the behaviour of storages with Middlewares. TinyDB provides Middlewares for caching and concurrency handling.
- **nearly 100% covered with tests**: If you don't count that `__repr__` methods and some abstract methods are not tested, TinyDB has a code coverage of 100%.

In short: If you need a simple database with a clean API that just works without lots of configuration, TinyDB might be the right choice for you.

1.1.1 Compatibility

TinyDB has been tested with Python 2.6, 2.7, 3.2, 3.3 and pypy.

1.2 Why not using TinyDB?

- You need **advanced features** like multiple indexes, an HTTP server, relationships, or similar.
- You are really concerned about **high performance** and need a high speed database.

To put it plainly: TinyDB is designed to be tiny and fun to use. If you need advanced features/high performance, TinyDB is the wrong database for you – consider using databases like Buzhug, CodernityDB or MongoDB.

CHAPTER 2

How to Use TinyDB

2.1 Basic Usage

2.1.1 Initializing

By default, TinyDB stores data as JSON files, so you have to specify the file path:

```
>>> from tinydb import TinyDB, where
>>> db = TinyDB('/path/to/db.json')
```

You also can use in-memory JSON to store data:

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

2.1.2 Inserting

As a document-oriented database, TinyDB uses `dicts` to store data:

```
>>> db.insert({'int': 1, 'char': 'a'})
>>> db.insert({'int': 1, 'char': 'b'})
>>> db.insert({'int': 1, 'value': 5.0})
```

2.1.3 Getting Data

Hint: Throughout this section, the return values are stripped of the `_id` key for clarity reasons. So:

```
[{'int': 1, 'value': 5.0}]
```

would actually be some something like:

```
[{'int': 1, 'value': 5.0, '_id': 3}]
```

Getting all data stored:

```
>>> db.all()
[{'int': 1, 'char': 'a'}, {'int': 1, 'char': 'b'}, {'int': 1, 'value': 5.0}]
```

Getting the database size (number of elements stored):

```
>>> len(db)
3
```

Search for all elements that have the `value` key defined:

```
>>> db.search(where('value'))
[{'int': 1, 'value': 5.0}]
```

Search for a specific value:

```
>>> db.search(where('int') == 1)
[{'int': 1, 'char': 'a'}, {'int': 1, 'char': 'b'}, {'int': 1, 'value': 5.0}]
```

Count the number of elements matching a query:

```
>>> db.count(where('int') == 1)
3
```

Check whether a specific element is in the database:

```
>>> db.contains(where('int') == 1)
True
>>> db.contains(where('int') == 0)
False
```

Alternative syntax for `db.contains`:

```
>>> if where('value') == 0.5 in db: print 'Found!'
Found!
```

Warning: Deprecated since version 1.3.0: This syntax will propably be removed soon. Please use the `db.contains(...)` syntax instead.

Other comparison operators you can use:

- `where('int') != 1`
- `where('int') < 2` and <=
- `where('int') > 0` and >=

Combine two queries with logical and:

```
>>> db.search((where('int') == 1) & (where('char') == 'b'))
[{'int': 1, 'char': 'b'}]
```

Combine two queries with logical or:

```
>>> db.search((where('char') == 'a') | (where('char') == 'b'))
[{'int': 1, 'char': 'a'}, {'int': 1, 'char': 'b'}]
```

When using `&` or `|`, make sure you wrap the conditions on both sides with parentheses or Python will mess up the comparison.

More advanced queries

Check against a regex:

```
>>> db.search(where('char').matches('[aZ]*'))
[{'int': 1, 'char': 'a'}, {'int': 1, 'char': 'b'}]
```

Use a custom test function:

```
>>> test_func = lambda c: c == 'a'
>>> db.search(where('char').test(test_func))
[{'char': 'a', 'int': 1}]
```

Also, if you want to get only one element, you can use:

```
>>> db.get(where('value'))
{'int': 1, 'value': 5.0}
```

> **Caution:** If multiple elements match the query, only one of them will be returned!

2.1.4 Removing

You can remove all elements matching a query:

```
>>> db.remove(where('int') == 1)
>>> len(db)
0
```

You also can purge all entries:

```
>>> db.purge()
>>> len(db)
0
```

2.1.5 Updating

You can update elements matching a query. Assuming you have these elements in the database:

```
>>> db.insert({'int': 1, 'char': 'a'})
>>> db.insert({'int': 1, 'char': 'b'})
>>> db.insert({'int': 1, 'value': 5.0})
```

Then you can update selected elements like this:

```
>>> db.update({'int': 2}, where('char') == 'a')
>>> db.all()
[{'int': 2, 'char': 'a'}, {'int': 1, 'char': 'b'}, {'int': 1, 'value': 5.0}]
```

2.2 Advanced Usage

2.2.1 Tables

You can use TinyDB with multiple tables. They behave exactly as described above:

```
>>> table = db.table('name')
>>> table.insert({'value': True})
>>> table.all()
[{'value': True}]
```

In addition, you can remove all tables by using:

```
>>> db.purge_tables()
```

Hint: When using the operations described above using `db`, TinyDB actually uses a table named `_default`.

2.2.2 Middlewares

Middlewares wrap around existing storages allowing you to customize their behaviour.

```
>>> from tinydb.storages import JSONStorage
>>> from tinydb.middlewares import CachingMiddleware
>>> db = TinyDB('/path/to/db.json', storage=CachingMiddleware(JSONStorage))
```

TinyDB ships with these middlewares:

- **CachingMiddleware**: Improves speed by reducing disk I/O. It caches all read operations and writes data to disk every `CachingMiddleware.WRITE_CACHE_SIZE` write operations.
- **ConcurrencyMiddleware**: Allows you to use TinyDB in multithreaded environments by using a lock on read and write operations, making them virtually atomic.

Hint: You can nest middlewares:

```
>>> from tinydb.middlewares import CachingMiddleware, ConcurrencyMiddleware
>>> db = TinyDB('/path/to/db.json', storage=ConcurrencyMiddleware(CachingMiddleware(JSONStorage)))
```

CHAPTER 3

How to Extend TinyDB

3.1 Storages

To write a custom storage, subclass `Storage`:

class `tinydb.storages.`**`Storage`**
The abstract base class for all Storages.

A Storage (de)serializes the current state of the database and stores it in some place (memory, file on disk, ...).

`read``()`
Read the last stored state.

Any kind of deserialization should go here.

Return type dict

`write``(`*`data`*`)`
Write the current state of the database to the storage.

Any kind of serialization should go here.

Parameters **data** (*dict*) – The current state of the database.

To use your custom storage, use:

```
db = TinyDB(storage=YourStorageClass)
```

Hint: TinyDB will pass all arguments and keyword arguments (except for `storage`) to your storage's `__init__`.

For example implementations, check out the source of `JSONStorage` or `MemoryStorage`.

3.2 Middlewares

To write a custom storage, subclass `Middleware`:

class `tinydb.middlewares.`**`Middleware`**
The base class for all Middlewares.

Middlewares hook into the read/write process of TinyDB allowing you to extend the behaviour by adding caching, logging, ...

Your middleware's `__init__` method has to accept exactly one argument which is the class of the "real" storage. It has to be stored as `_storage_cls` (see `CachingMiddleware` for an example).

storage
This will contain the underlying `Storage`

read(*self*)
Modify the way TinyDB reads data.

To access the underlying storage's read method, use `self.storage.read`.

write(*self*, *data*)
Modify the way TinyDB writes data.

To access the underlying storage's read method, use `self.storage.read`.

To use your middleware, use:

```
db = TinyDB(storage=YourMiddleware(SomeStorageClass))
```

For example implementations, check out the source of `CachingMiddleware` or `ConcurrencyMiddleware`.

CHAPTER 4

Limitations of TinyDB

4.1 JSON Serialization

TinyDB serializes all data using the Python JSON module by default. It serializes most basic Python data types very well, but fails serializing classes and stores `tuple` as `list`. If you need a better serializer, you can write your own storage, that e.g. uses the more powerful (but also slower) pickle or PyYAML.

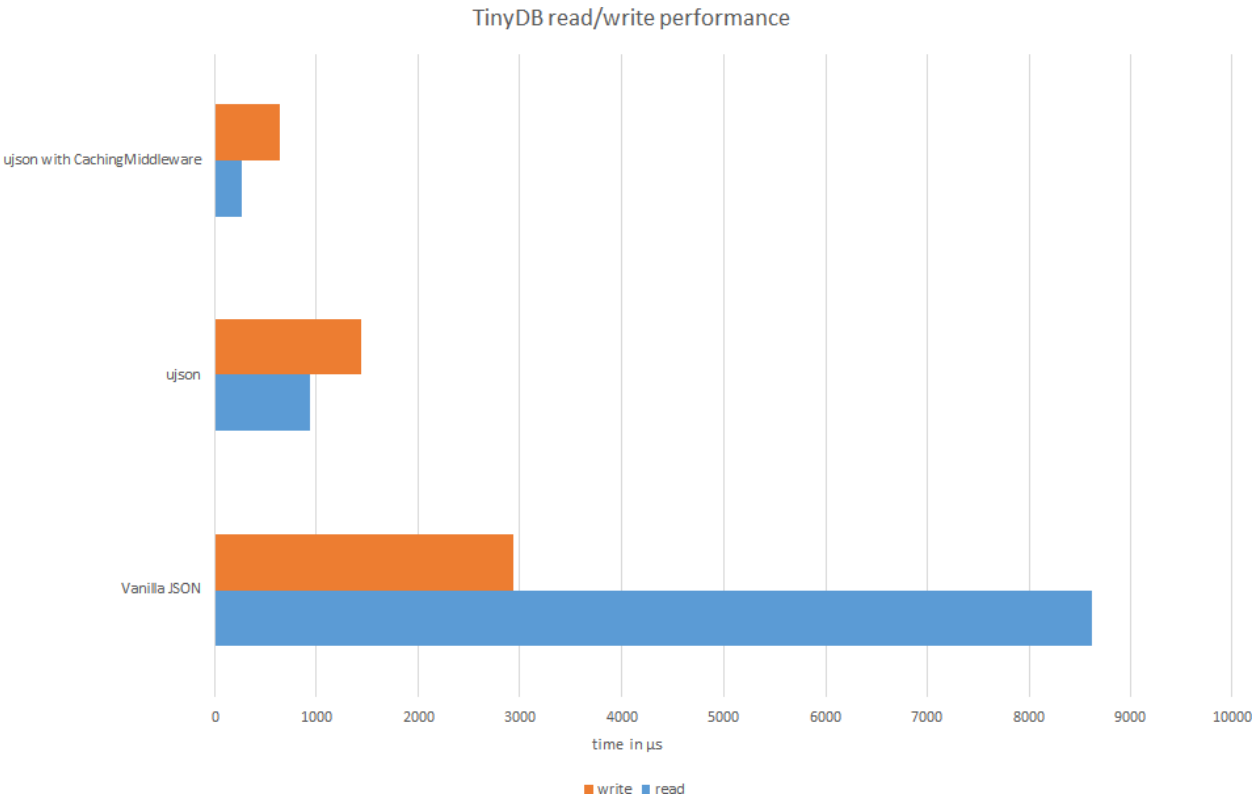
4.2 Performance

TinyDB is NOT designed to be used in environments where performance might be an issue. Although you can improve the TinyDB performance as described below, you should consider using a DB that is optimized for speed like Buzhug or CodernityDB.

4.2.1 How to Improve TinyDB Performance

The default storage serializes the data using JSON. To improve performance, you can install ujson , an extremely fast JSON implementation. TinyDB will auto-detect and use it if possible.

In addition, you can wrap the storage with the `CachingMiddleware` which reduces disk I/O.


TinyDB read/write performance
ujson with CachingMiddleware
ujson
Vanilla JSON
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
time in µs
write
read

CHAPTER 5

API Documentation

5.1 tinydb.database

Contains the database and tables implementation.

class tinydb.database.**Table**(*name*, *db*)
Represents a single TinyDB Table.

__contains__(*condition*)
Equals to bool(table.search(condition))).

__enter__()
Allow the database to be used as a context manager.

Returns the table instance

__exit__(**args*)
Try to close the storage after being used as a context manager.

__init__(*name*, *db*)
Get access to a table.

Parameters

- **name** (*str*) – The name of the table.
- **db** (*tinydb.database.TinyDB*) – The parent database.

__len__()
Get the total number of elements in the table.

all()
Get all elements stored in the table.

Note: all elements will have an *_id* key.

Returns a list with all elements.

Return type list[dict]

contains(*cond*)
Check wether the database contains an element matching a condition.

Parameters **cond** (*Query*) – the condition use

count(*cond*)
Count the elements matching a condition.

Parameters **cond** (*Query*) – the condition use

get(*cond*)
Search for exactly one element matching a condition.

Note: all elements will have an *_id* key.

Parameters **cond** (*Query*) – the condition to check against

Returns the element or None

Return type dict or None

insert(*element*)
Insert a new element into the table.

element has to be a dict, not containing the key 'id'.

purge()
Purge the table by removing all elements.

remove(*cond*)
Remove the element matching the condition.

Parameters **cond** (*query, int, list*) – the condition to check against

search(*cond*)
Search for all elements matching a 'where' cond.

Note: all elements will have an *_id* key.

Parameters **cond** (*Query*) – the condition to check against

Returns list of matching elements

Return type list

update(*fields*, *cond*)
Update all elements matching the condition to have a given set of fields.

Parameters

- **fields** (*dict*) – the fields that the matching elements will have
- **cond** (*query*) – which elements to update

class tinydb.database.**TinyDB**(**args*, ***kwargs*)
The main class of TinyDB.

Gives access to the database, provides methods to insert/search/remove and getting tables.

__contains__(*item*)
A shorthand for `query(...) == ... in db.table()`. Intendet to be used in if-clauses (avoiding `if len(db.serach(...)):`)

```
>>> if where('field') == 'value' in db:
...     print True
```

__enter__()
See `Table.__enter__()`

__exit__(**args*)
See `Table.__exit__()`

__getattr__(*name*)
Forward all unknown attribute calls to the underlying standard table.

__init__(**args*, ***kwargs*)
Create a new instance of TinyDB.

All arguments and keyword arguments will be passed to the underlying storage class (default: `JSONStorage`).

__len__()
Get the total number of elements in the DB.

```
>>> len(db)
0
```

purge_tables()
Purge all tables from the database. **CANT BE REVERSED!**

table(*name='_default'*)
Get access to a specific table.

Creates a new table, if it hasn't been created before, otherwise it returns the cached `Table` object.

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

5.2 tinydb.queries

Contains the querying interface.

Starting with `Query` you can construct complex queries:

```
>>> ((where('f1') == 5) & (where('f2') != 2)) | where('s').matches('^\w+$')
(('f1' == 5) and ('f2' != 2)) or ('s' ~= ^\w+$ )
```

Queries are executed by using the `__call__`:

```
>>> q = where('val') == 5
>>> q({'val': 5})
True
>>> q({'val': 1})
False
```

class tinydb.queries.**Query**(*key*)
Provides methods to do tests on dict fields.

Any type of comparison will be called in this class. In addition, it is aliased to `where` to provide a more intuitive syntax.

When not using any comparison operation, this simply tests for existence of the given key.

__call__(*element*)
Run the test on the element.

Parameters **element** (*dict*) – The dict that we will run our tests against.

__eq__(*other*)
Test a dict value for equality.

```
>>> where('f1') == 42
'f1' == 42
```

__ge__(*other*)
Test a dict value for being greater than or equal to another value.

```
>>> where('f1') >= 42
'f1' >= 42
```

__gt__(*other*)
Test a dict value for being greater than another value.

```
>>> where('f1') > 42
'f1' > 42
```

__invert__()
Negates a query.

```
>>> ~(where('f1') >= 42)
not ('f1' >= 42)
```

Return type tinydb.queries.QueryNot

__le__(*other*)
Test a dict value for being lower than or equal to another value.

```
>>> where('f1') <= 42
'f1' <= 42
```

__lt__(*other*)
Test a dict value for being lower than another value.

```
>>> where('f1') < 42
'f1' < 42
```

__ne__(*other*)
Test a dict value for inequality.

```
>>> where('f1') != 42
'f1' != 42
```

matches(*regex*)
Run a regex test against a dict value.

```
>>> where('f1').matches('^\w+$')
'f1' ~= ^\w+$
```

Parameters **regex** – The regular expression to pass to `re.match`

Return type QueryRegex

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

```
>>> def test_func(val):
...     return val == 42
...
>>> where('f1').test(test_func)
'f1'.test(<function test_func at 0x029950F0>)
```

Parameters **func** – The function to run. Has to accept one parameter and return a boolean.

Return type QueryCustom

class tinydb.queries.**AndOrMixin**
A mixin providing methods calls & and |.

All queries can be combined with & and |. Thus, we provide a mixin here to prevent repeating this code all the time.

__and__(*other*)
Combines this query and another with logical and.

Example:

```
>>> (where('f1') == 5) & (where('f2') != 2)
('f1' == 5) and ('f2' != 2)
```

Return type QueryAnd

__or__(*other*)
Combines this query and another with logical or.

Example:

```
>>> (where('f1') == 5) | (where('f2') != 2)
('f1' == 5) or ('f2' != 2)
```

Return type QueryOr

5.3 tinydb.storage

Contains the base class for storages and two implementations.

class tinydb.storages.**JSONStorage**(*path*)
Store the data in a JSON file.

__init__(*path*)
Create a new instance.

Also creates the storage file, if it doesn't exist.

Parameters **path** (*str*) – Where to store the JSON data.

class tinydb.storages.**MemoryStorage**
Store the data as JSON in memory.

__init__()
Create a new instance.

5.4 tinydb.middlewares

Contains the base class for middlewares and two implementations.

class tinydb.middlewares.**CachingMiddleware**(*storage_cls*)
Add some caching to TinyDB.

This Middleware aims to improve the performance of TinyDB by writing only the last DB state every WRITE_CACHE_SIZE time and reading always from cache.

flush()
Flush all unwritten data to disk.

class tinydb.middlewares.**ConcurrencyMiddleware**(*storage_cls*)
Makes TinyDB working with multithreading.

Uses a lock so write/read operations are virtually atomic.

CHAPTER 6

Changelog

6.1 Version Numbering

TinyDB follows the SemVer versioning guidelines. For more information, see semver.org

6.2 v1.3.0 (2014-07-02)

- Fixed bug #7: IDs not unique.
- Extended the API: `db.count(where(...))` and `db.contains(where(...))`
- The syntax `where(...) in db` is now **deprecated** and replaced by `db.contains`.

6.3 v1.2.0 (2014-06-19)

- Added `update` method (see Issue #6).

6.4 v1.1.1 (2014-06-14)

- Merged PR #5: Fix minor documentation typos and style issues.

6.5 v1.1.0 (2014-05-06)

- Improved the docs and fixed some typos.
- Refactored some internal code.
- Fixed a bug with multiple `TinyDB?` instances.

6.6 v1.0.1 (2014-04-26)

- Fixed a bug in `JSONStorage` that broke the database when removing entries.

6.7 v1.0.0 (2013-07-20)

- First official release – consider TinyDB stable now.

t