EssentialDB Documentation

Release 0.4

Shane C Mason

Oct 05, 2018

Contents

CHAPTER 1

Quickstart

Install with pip:

```
pip install essentialdb
```

Basic usage is straightforward:

```
from essentialdb import EssentialDB

#create or open the database
db = EssentialDB(filepath="my.db")

#get the collection
authors = db.get_collection('authors')

#use the abbreviated syntax
books = db.books

#insert a document into the collection
authors.insert_one({'first': 'Langston', 'last': 'Hughes', 'born': 1902});

#find some entries
results = authors.find({'last':'Hughes'})

#commit the changes to disk
db.sync()
```

Or using the 'with' semantics to assure that write happen without having to explicitly call sync:

```
with db.authors as author_collection:
    author_collection.insert_one({'first': 'Langston', 'last': 'Hughes', 'born': 1902}
↪)
```

Insert a document:

```
author = {'first': 'Langston', 'last': 'Hughes', 'born': 1902}
author_collection.insert_one(author)
```

Insert many documents:

```
authors = [{'first': 'Langston', 'last': 'Hughes', 'born': 1902},
           {'first': 'Ezra', 'last': 'Pound', 'born': 1885}]
author_collection.insert_many(authors)
```

Find one document:

```
document = author_collection.find_one({'first': 'Ezra', 'last': 'Pound'})
```

Find many:

```
documents = author_collection.find({'born': {'$gt': 1900}})
```

Update one:

```
updated = author_collection.update({'_id': {'$eq': "A345i"}}, {'born': 1902})
```

Update many:

```
updated = author_collection.update({'born': {'$gt': 1900}}, {'period': 'Modern'})
```

Note that updating fields that don't currently exist in the document will add the field to the document.

Remove Items:

```
removed = author_collection.remove({'period':'Modern'})
```

Nested Queries:

```
customer_collection.insert_one({'first': 'John', 'last': 'Smith', 'address': { 'street
↪': '10 Maple St', 'city': 'Missoula', 'state': 'MT'}})
results = customer_db.find({'address.state':'MT'})
```

Note that nested query support means that key names can not include a period.

Write updates to disk:

```
author_collection.sync()
```

CHAPTER 2

Queries

Queries in EssentialDB follow the same basic form as MongoDB:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

2.1 Comparison Query Selectors

The $eq operator matches documents where the value of a field equals the specified value:

```
author_collection.find({"born" : {"$eq": 1972}})
```

The $ne operator matches documents where the value of a field is not equal to the specified value:

```
author_collection.find({"born" : {"$ne": 1972}})
```

The $gt operator matches documents where the value of a field is greater than the specified value:

```
author_collection.find({"born" : {"$gt": 1900}})
```

The $gte operator matches documents where the value of a field is great than or equal to the specified value:

```
author_collection.find({"born" : {"$gte": 1900}})
```

The $lt operator matches documents where the value of a field is less than the specified value:

```
author_collection.find({"born" : {"$lt": 1900}})
```

The $lte operator matches documents where the value of a field is less than or equal to the specified value:

```
author_collection.find({"born" : {"$lte": 1900}})
```

The $in operator matches documents where the value of a field is equal any item in the specified array:

```
author_collection.find({"genre" : {"$in": ["tragedy", "drama"]}})
```

The $nin operator matches documents where the value of a field is not equal to any item in the specified array:

```
author_collection.find({"genre" : {"$nin": ["tragedy", "drama"]}})
```

2.2 Boolean Operators

The $and operator matches documents where all the fields match:

```
#find authors born after 1900 and before 2000
author_collection.find({'$and':[{'born': {'$gte': 1900}},{'born': {'$lt': 2000}}]})
```

The $or operator matches documents where any of the fields match:

```
#find authors with either the first or last name John
author_collection.find({'$or':[{'first': {'$eq': 'John'}},{'last': {'$eq': 'John'}}]})
```

The $nor operator matches document where none of the conditions match:

```
#find all authors who have neither the first or last name John
author_collection.find({"$nor":[{'first': {"$eq": 'John'}},{'last': {'$eq': 'John'}}]}
↪)
```

CHAPTER 3

Serializers

EssentialDB does not currently support any partitioning or sharding schemes, so data is written and loaded to disk at once. This is okay, since EssentialDB is designed to be used mostly in-memory, but it means that disk writes are expensive. We did some fairly extensive benchmarking of Python 3 serializers and found that Pickle is the fastest and most versatile serialization techniques and is the default mechanism in EssentialDB for (de)serializing documents on disk.

Recognizing there are plenty of times when another format might be desired, version 0.5 adds support for Python's built-in JSON (de)serializer by adding the 'serializer' flag. You can invoke like this:

```
import essentialdb
db =  essentialdb.EssentialDB(filepath=self.path, serializer=essentialdb.
↪JSONSerializer())
```

What if you want to use something else though? Like ujson or msg-pack? Simply define your own storage class that implement the load and dump methods:

```
import msgpack
import essentialDB

class MsgPackSerializer:
    """
    Implements a basic (de)serializer based on MessagePack.
    """

    @staticmethod
    def load(file_path):
        with open(file_path, 'rb') as fp:
            data = msgpack.unpack(fp, encoding='utf-8')
        return data

    @staticmethod
    def dump(data, file_path):
        with open(file_path, 'wb') as fp:
            msgpack.pack(output, fp, use_bin_type=True)
```

(continues on next page)

(continued from previous page)

```
db =  essentialdb.EssentialDB(filepath=self.path, serializer=essentialdb.
↪MsgPackSerializer())
```

Beaware though, serializers not based on pickle will likely perform slower in most use cases. Assitionally, they will not natively support many Python types (like datetime). You can add those to most serializers with custom hooks. EssentialDB does not include native support for non-builtin (de)serializers to avoid external dependencies.

CHAPTER 4

Performance

The purpose of this document is to describe the performance characteristics of EssentialDB over a range of data sizes. Since the purpose of EssentialDB is to aid speed of development in the early stages while allowing the project to scale, I wanted the test to have a real world flavor.

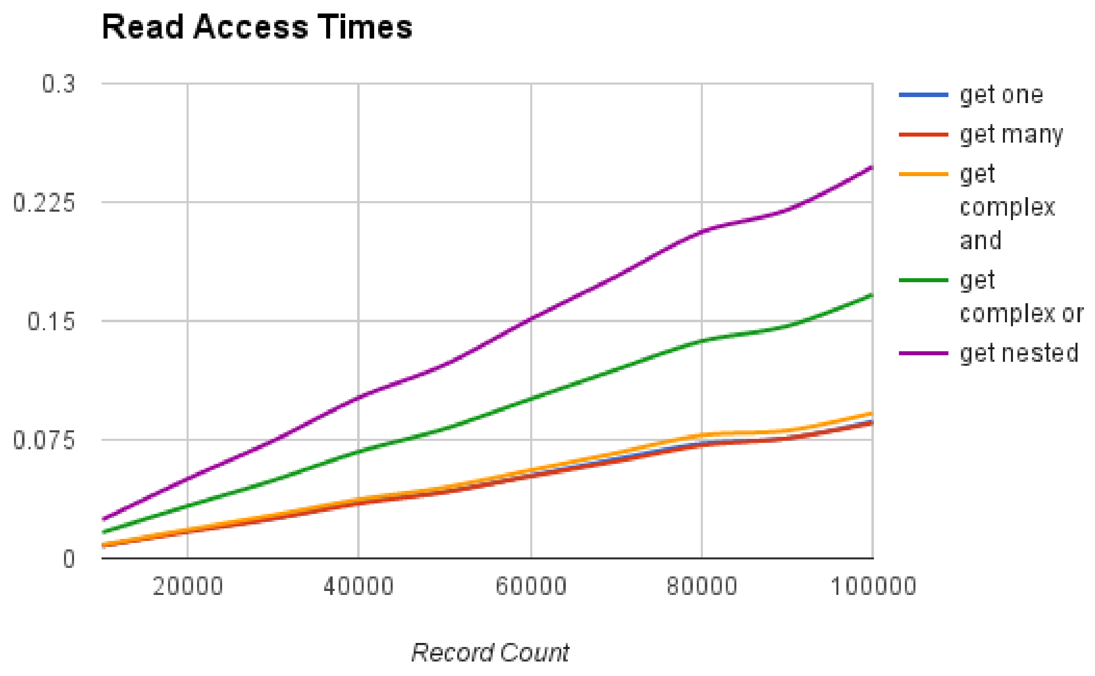
Think about this scenario:

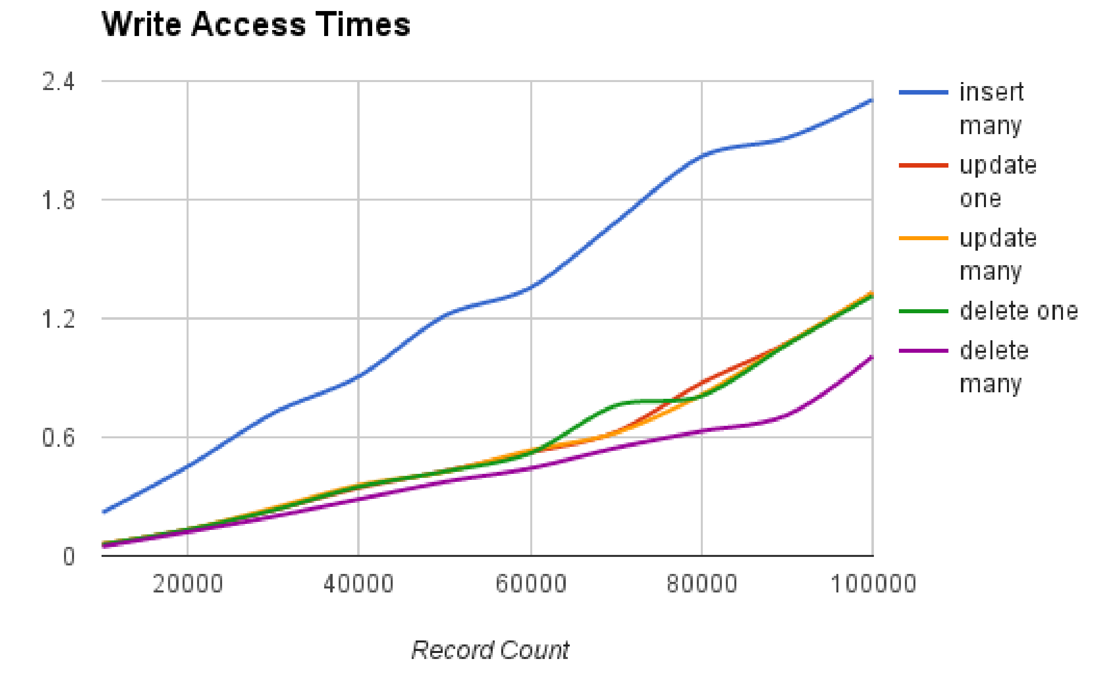
> Your team has a short fuse program kick off to create an in-house ticket tracking system ASAP. Initially, it will only be used by the local office, but if it tests well your manager is hoping to open it up to other branches over the next year. At the kick off meeting, she announces that she is even hoping to get corporate to adopt it if it tests well.

We've all been here. Speed of development is the most pressing factor, but scalability matters too.

> Some quick napkin sketches tell you that each branch will add around a hundred new users, and new projects and whole new use cases. If you pull this off, it could be a big deal for your career, but the first milestones are just weeks away…

If you choose EssentialDB to kickstart your development, we want to make sure you understand exactly how it will scale. The figures below illustrate how EssentialDB scales in this use case, while the dataset grows from 10k to 100k. These test were ran on generated mock data designed to model a growing database for a ticket tracking system. I ran them on a standard AWS T2.medium. The methodology is covered in detail in the rest of the article.





What do these graphs tell us about EssentialDB? In this scenario, it scaled linearly and reads are much more efficent than writes. For our use case, it looks like a pretty fair solution at the early stages, to demonstrate progress quickly and see if the product catches on. It a good thing that all your queries should work for MongoDB right out the box!

4.1 The Data

The first requirement here is that tests could be replicated - this required consistent data. For this, we generated all the data up front. The documents are based on the following example:

```
{
    "permissions":{
        "owner": "randomuser@example.com",
        "contributors": ["rand1@example.com", "rand2@example.com" ...]
    },
    "slug": "a_unique_string",
    "created": 1470455125.213304,
    "due": 1474083925.213304,
    "priority": "minor",
    "status": "started",
    "projects": ["project1", "project2" ... ],
    "labels": ["python", "database" ... ],
    "title": "Test Python Databases",
    "details": "Test several python databases head to head."
 }
```

I created a script to generate random data and save it to disk as plain old python objects (dictionaries really) using shelve. This way, the data can be read into the different databases so that results are constant. The script generates each data entry using the following rules:

- **permissions** - Complex object
- **permissions.owner** - Random fake email address from users list
- **permissions.contributers** - Array of 0 to 3 fake email addresses from users list
- **slug** - A unique string based on the project name and incrementing counts
- **created** - Integer epoch timestamp of creation (1 to 500 days in the past)
- **due** - Integer epoch timestamp the task is due (0 to 30 days after created timestamp)
- **priority** - randomly selected from [none, minor, major, critical, blocker]
- **status** - randomly selected from [none, started, blocked, closed, completed]
- **projects** - Array of 1 to 4 words, randomly selected from a list of generated project 'names'
- **labels** - Array of 0 to 3 labels, randomly selected from a list of generated labels
- **title** - String 4 to 12 random lorem ipsum words
- **description** - String of 1 and 10 random lorem ipsum paragraphs

The script begins with 10k records and then adds 10k records until it reaches 100k, saving each iteration to disk [100k, 200k, 300k … 1000k]. To model growth you might see in a real system, for each 100 k records added, the script generates 1000 additional users, project names and labels to be randomly selected when generating new entries. After the new entries are added to the dataset, the script shuffles before writing to disk.

With 10 files ranging from 10K to 100K - modeling snapshots of a growing database - we are ready to start looking at the tests.

4.2 The Tests

I created a timing script <LINK HERE> to run through each of the previously stored datasets. Basically, it operates list this

```
For dataset in [10k, 20k ... 100k]  populate database For each function
        For each iteration
            initialize database object start timer run function stop timer
          average = time/iterations
```

4.2.1 Functions

- **insert_multiple** - Adds a list of documents into the database
- **get_one** - Queries for one document by slug
- **get_many** - Query for all documents where priority == minor
- **get_complex_and** - Query for all documents where priority == blocker AND status == blocked
- **get_complex_or** - Query for all documents where status == closed OR priority == critical
- **get_nested** - Query for all documents where permissions.owner == prechosen name
- **update_one** - Update one document to have status = completed
- **update_many** - Update all where priority == minor to have status = closed
- **delete_one** - Delete one document by slug
- **delete_many** - Delete all documents where status == none

These tests were run on an Amazon AWS t2.medium instance.

4.3 The Results

The following table shows the outcome of read operations.

Records	get one	get many	get complex and	get complex or	get nested
10000	0.008	0.008	0.009	0.016	0.024
20000	0.017	0.017	0.018	0.033	0.050
30000	0.026	0.025	0.027	0.049	0.074
40000	0.035	0.035	0.037	0.067	0.102
50000	0.042	0.042	0.045	0.082	0.122
60000	0.052	0.052	0.056	0.101	0.151
70000	0.063	0.061	0.067	0.119	0.178
80000	0.072	0.071	0.078	0.137	0.206
90000	0.076	0.076	0.081	0.147	0.220
100000	0.086	0.085	0.092	0.167	0.247

The following table shows the outcome of write operations.

4.4 Analysis

Currently, performance is pretty good for a pure python database. In this scenario, it scaled linearly and reads are much more efficient than writes. There are many use cases where it's current performance would be suitable and many where it would not be.

CHAPTER 5

Collection API

class essentialdb.**Collection**(*documents*, *threading_lock*, *onsync_callback*, *autosync=False*, *name=None*)

count()
Get the total number of documents in the collection.

createIndex(*index*, *options=None*)
Create an index.

dropIndexes()
Drop all indexes.

find(*query={}*, *filter=None*)
Finds all documents that match the query. If query is not specified, all documents in the collection will be returned.

Kwargs: query (dict): If specified, the query to be ran on the collection. filter (fund): If specified, results will be ran through provided function before inclustion in results

Returns: A list of matching documents or None if no documents match query or the collection is empty.

Example:

```
with library_db.authors as author_collection:
    document = author_collection.find({'last': 'Smith'})
```

find_one(*query=None*, *filter=None*)
Finds one document that matches the query. If multiple documents match (or query is not specified), a single random document is returned from the result set.

Kwargs: query (dict): If specified, the query to be ran on the collection.

Returns A single matching document or None if no documents match query or the collection is empty.

Example:

```
with library_db.authors as author_collection:
    document = author_collection.find_one({'first': 'Ezra', 'last': 'Pound'})
```

get (*key*)

Get a document previously inserted by 'set' or any document whose _id is equal to 'key'. This is a short-cut function designed designed to provide EssentialDB with semantics similar to a transparent key/value store (like redis).

Args: key (string) The key (_id) of the item to be retrieved.

Returns: The document (dict) if found.

Example:

```
with my_db.cache as request_cache:
    response.text = request_cache.get( request.url )
```

insert_many (*documents*)

Inserts a list of documents into the collection using the same process as oulined for insert_one

Args: documents (list) A list of documents (dict) to insert.

Returns The number of documents inserted into the store.

Example:

```
with library_db.authors as author_collection:
    authors = [{'first': 'Langston', 'last': 'Hughes', 'born': 1902},
    {'first': 'Ezra', 'last': 'Pound', 'born': 1885}]
    author_collection.insert_many(authors)
```

insert_one (*document*)

Inserts one document into the collection. If the document already contains an _id, or one is specified in the key word arguments, it will be used as the documents primary identifier on insert. If a document with the same identifier already exists, this operation will overwrite the existing document. If '_id' is not specified, a globally unique identifier will be generated and inserted.

Args: document (dict) The document to insert.

Returns: The unique identifier for the inserted document

Example:

```
with library_db.authors as author_collection:
    #documents are just dictionaries
    author = {'first': 'Langston', 'last': 'Hughes', 'born': 1902}
    author_collection.insert_one(author)
```

remove (*query=None*)

Remove all documents that match query. If query is not specified, all documents will be removed.

Kwargs: query (dict): The query to be ran on the collection. An empty dictionary {} matches all.

Returns: The number of documents removed.

Example:

```
with library_db.books as book_collection:
    document = book_collection.remove({'period': 'Modern'})
```

set (*key*, *value*)

Sets key in the current collection to be equal to the provided value. This is a short-cut function designed designed to provide EssentialDB with semantics similar to a transparent key/value store (like redis). Under the hood, this is creating a document from 'value' and assigning it the _id of 'key'. Any additional sets will overwrite the current value.

Args: key (string) The key for the value to be inserted. value (dict) The value to set.

Returns: The unique identifier for the inserted document

Example::

with my_db.cache as request_cache: request_cache.set(request.url, response.text)

update (*query*, *update*)

Applies the specified update to all documents in the collection that match the specified query.

Args: query (dict): The query to be ran on the collection. An empty dictionary {} matches all. update (dict): The update to be ran on matching documents.

Returns: The number of documents updated.

Example:

```
with library_db.books as book_collection:
    updated = book_collection.update({'year': {'$gt': 1900}}, {'period':
↪'Modern'})
```

CHAPTER 6

EssentialDB

For Python 3.4, 3.5, 3.6 & 3.7(Nightly)

EssentialDB is a pure Python document database developed to meet the following tenets:

1. Databases shouldn't slow down development - a developer should be able to integrate a database in less than a minute.
2. Development databases should have a complete feature set and be performant enough for prototyping and project startups
3. Development databases should provide a path to scale when your project takes off

Project On GitHub | Full Docs @ ReadTheDocs | Distribution On Pypi

6.1 Speeding Development

Our first tenet is that you should be able to start developing in less than a minute. Since EssentialDB is an 'embedded' database, there is no external services or dependencies to install or administrate. The tenet here is to take you from concept to development in less than a minute. Install with pip:

```
pip install essentialdb
```

Basic usage is straightforward:

```
from essentialdb import EssentialDB

#create or open the database
db = EssentialDB(filepath="my.db")

#get the collection
authors = db.get_collection('authors')

#use the abbreviated syntax
```

(continues on next page)

(continued from previous page)

```
books = db.books

#insert a document into the collection
authors.insert_one({'first': 'Langston', 'last': 'Hughes', 'born': 1902});

#find some entries
results = authors.find({'last':'Hughes'})

#commit the changes to disk
db.sync()
```

Or using the 'with' semantics to assure that write happen without having to explicitly call sync:

```
with db.authors as author_collection:
    author_collection.insert_one({'first': 'Langston', 'last': 'Hughes', 'born': 1902}
↪)
```

Insert a document:

```
author = {'first': 'Langston', 'last': 'Hughes', 'born': 1902}
author_collection.insert_one(author)
```

Insert many documents:

```
authors = [{'first': 'Langston', 'last': 'Hughes', 'born': 1902},
           {'first': 'Ezra', 'last': 'Pound', 'born': 1885}]
author_collection.insert_many(authors)
```

Find one document:

```
document = author_collection.find_one({'first': 'Ezra', 'last': 'Pound'})
```

Find many:

```
documents = author_collection.find({'born': {'$gt': 1900}})
```

Update one:

```
updated = author_collection.update({'_id': {'$eq': "A345i"}}, {'born': 1902})
```

Update many:

```
updated = author_collection.update({'born': {'$gt': 1900}}, {'period': 'Modern'})
```

Note that updating fields that don't currently exist in the document will add the field to the document.

Remove Items:

```
removed = author_collection.remove({'period':'Modern'})
```

Nested Queries:

```
customer_collection.insert_one({'first': 'John', 'last': 'Smith', 'address': { 'street
↪': '10 Maple St', 'city': 'Missoula', 'state': 'MT'}})
results = customer_db.find({'address.state':'MT'})
```

Note that nested query support means that key names can not include a period.

Write updates to disk:

```
author_collection.sync()
```

6.2 Features & Performance

Our second tenet is that EssentialDB should have the performance and features you need to get your project rolling.

EssentialDB supports a very rich queries that follow the same basic form as MongoDB:

```
{ <field1>:  <operator1>: <value1> }, ... }
```

Most comparison operators are supported, including equals, not equals, less than, greater than:

```
author_collection.find({"born" : {"$gt": 1900}})
```

You can even test against lists of items using $in and $nin:

```
author_collection.find({"genre" : {"$in": ["tragedy", "drama"]}})
```

AND and OR boolean operators allow you to make arbitrarily complex queries:

```
#find authors born after 1900 and before 2000
author_collection.find({'$and':[{'born': {'$gte': 1900}},{'born': {'$lt': 2000}}]})

#find authors with either the first or last name John
author_collection.find({'$or':[{'first': {'$eg': 'John'}},{'last': {'$eq': 'John'}}]})
```

We've tested EssentialDB under some typical use cases, and seen that it is plenty performant for many use cases with small to moderate loads.

6.3 Where is it used?

EssentialDB is being used in a variety of small projects. Most notably, it is powering some of the features behind kinder.farm.

Changelog:

- **0.5 - 01.07.2017 - Several changes, including::**
  - Added thread lock for inserts, updates and disk writes
  - Refactored serialization into PickSerializer class
  - Added JSONSerializer class
  - Added support for custom serializers
  - Renamed SimpleCollection to LocalCollection
- 0.4 - 12.30.2016 - Huge performance improvements from removing SimpleDocument dict wrapper
- 0.3 - 12.28.2016 - Added hashed indexes and documentation updates
- 0.2 - 12.18.2016 - Complete rewrite of query system to now precompile the query document.

Index