semidbm Documentation

Release 0.4.0

James Saryerwinnie Jr

September 04, 2013

CONTENTS

semidbm is a pure python implementation of a dbm, which is essentially a persistent key value store. It allows you to get and set keys:

```
import semidbm
db = semidbm.open('testdb', 'c')
db['foo'] = 'bar'
print db['foo']

db.close()
# Then at a later time:
db = semidbm.open('testdb', 'r')
# prints "bar"
print db['foo']
```

It was written with these things in mind:

- Pure python. Many of the "standard" dbms are written in C, which requires a C extension to give python code access to the dbm. This of course make installation more complicated, requires that the OS you're using has the shared library installed, requires that you have the necessary setup to build a C extension for python (unless you want to use a binary package). Semidbm supports python 2.6, python 2.7, and python 3.3.
- Cross platform. Because semidbm is written in python, it runs on any platform that supports python. The file format used for semidbm is also cross platform.
- Simplicity. The original design is based off of python's dumbdbm module in the standard library, and one of the goals of semidbm is to try to keep the design comparably simple.

Post feedback and issues on github issues, or check out the latest changes at the github repo.

CHAPTER

ONE

TOPICS

1.1 An Overview of Semidbm

The easiest way to think of semidbm is as an improvement over python's dumbdbm module.

While the standard library has faster dbms based on well established C libraries (GNU dbm, Berkeley DB, ndbm), dumbdbm is the only pure python portable dbm in the standard library.

Semidbm offers a few improvements over dumbdbm including:

- Better overall performance (more on this later).
- Only a single file is used (no separate index and data file).
- Data file compaction. Free space can be reclaimed (though this only happens whenever explicitly asked to do so using the *compact()* method).
- Get/set/delete are require O(1) IO.

Like dumbdbm, semidbm is cross platform. It has been tested on:

- Linux (Ubuntu 11.10, debian)
- Mac OS X (Lion/Mountain Lion)
- Windows 7/8.

There are also a few caveats to consider when using semidbm:

- The entire index must fit in memory, this means all keys must fit in memory.
- Not thread safe; can only be accessed by a single process.
- While the performance is reasonable, it still will not beat one of the standard dbms (GNU dbm, Berkeley DB, etc).

1.1.1 Using Semidbm

To create a new db, specify the name of the directory:

```
import semidbm
db = semidbm.open('mydb', 'c')
```

This will create a *mydb* directory. This directory is where semidbm will place all the files it needs to keep track of the keys and values stored in the db. If the directory does not exist, it will be created.

Once the db has been created, you can get and set values:

```
db['key1'] = 'value1'
print db['key1']
```

Keys and values can be either str or bytes.

`str` types will be encoded to utf-8 before writing to disk. You can avoid this encoding step by providing a byte string directly:

```
db[b'key1'] = b'value1'
```

Otherwise, semidbm will do the equivalent of:

```
db['key1'.encode('utf-8')] = 'value1'.encode('utf-8')
```

It is recommended that you handle the encoding of your strings in your application, and only use `bytes` when working with semidbm. The reason for this is that when a value is retrieved, it is returned as a bytestring (semidbm can't know the encoding of the bytes it retrieved). For example (this is with python 3.3):

```
>>> db['foo'] = 'value'
>>> db['foo']
b'value'
>>> db['ky'] = 'value'
>>> db['ky']
b'value\xc4\x93'
```

To avoid this confusion, encode your strings before storing with with semidbm.

The reason this automatic conversion is supported is that this is what is done with the DBMs in the python standard library (including `dumbdbm` which this module was intended to be a drop in replacement for). In order to be able to be a drop in replacement, this automatic encoding process needs to be supported (but not recommended).

1.2 SemiDBM Details

This guide goes into the details of how semidbm works.

1.2.1 Writing a Value

One of the key characteristics of semidbm is that it only writes to the end of a file. **Once data has been written to a file, it is never changed.** This makes it easy to guarantee that once the data is written to disk, you can be certain that semidbm will not corrupt the data. This also makes semidbm simpler because we don't have to worry about how to modify data in a way that prevents corruption in the event of a crash.

Even updates to existing values are written as new values at the end of a file. When the data file is loaded, these transactions are "replayed" so that the last change will "win". For example, given these operations:

```
add key "foo" with value "bar"
add key "foo2" with value "bar2"
delete key "foo2"
add key "foo" with value "new value"
```

this would represent a dictionary that looked like this:

```
{"foo": "new value"}
```

Note: This is just the conceptual view of the transactions. The actual format is a binary format specified in *File Format of DB file*.

You can imagine that a db with a large number of updates can cause the file to grow to a much larger size than is needed. To reclaim fixed space, you can use the `compact()` method. This will rewrite the data file is the shortest amount of transactions needed. The above example can be compacted to:

```
add key "foo" with value "new value"
```

When a compaction occurs, a new data file is written out (the original data file is left untouched). Once all the compacted data has been written out to the new data file (and fsync'd!), the new data file is renamed over the original data file, completing the compaction. This way, if a crash occurs during compaction, the original data file is not corrupted.

> **Warning:** In Windows, it is not possible to rename a file over an open file without using the low level win32 API directly. This can certainly be done, but the workaround for now is (only on windows) to rename the current file to a different filename, then to rename to new data file to the current file name, and finally to delete the original data file.

1.2.2 Reading Values

The index that is stored in memory does not contain the actual data associated with the key. Instead, it contains the location within the file where the value is located, conceptually:

```
db = {'foo': DiskLocation(offset=40, size=10)}
```

When the value for a key is requested, the offset and size are looked up. A disk seek is performed and a read is performed for the specified size associated with the value. This translates to 2 syscalls:

```
lseek(fd, offset, os.SEEKSET)
data = read(fs, value_size)
```

1.2.3 Data Verification

Every write to a semidbm db file also includes a crc32 checksum. When a value is read from disk, semidbm can verify this crc32 checksum. By default, this verification step is turned off, but can be enabled using the `verify_checksums` argument:

```
>>> db = semidbm.open('dbname', 'c', verify_checksums=True)
```

If a checksum error is detected a `DBMChecksumError` is raised:

```
>>> db[b'foo']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./semidbm/db.py", line 192, in __getitem__
    return self._verify_checksum_data(key, data)
  File "./semidbm/db.py", line 203, in _verify_checksum_data
    "Corrupt data detected: invalid checksum for key %s" % key)
semidbm.db.DBMChecksumError: Corrupt data detected: invalid checksum for key b'foo'
```

1.2.4 Read Only Mode

SemiDBM includes an optimized read only mode. If you know you only want to read values from the database without writing new values you can take advantage of this optimized read only mode. To open a db file as read only, use the `'r'` option:

```
db = semidbm.open('dbname', 'r')
```

When this happens, the data file is mmap'd. Instead of using an `lseek/read` to retrieve data files, we can just return the contents directly:

```
return contents[offset:offset+size]
```

This avoids a copying the contents to an additional buffer.

1.3 Benchmarking Semidbm

Semidbm was not written to be the fastest dbm available, but its performance is surprisingly well for a pure python dbm. Before showing the benchmark results, it's important to note that these benchmark results can vary across machines and should in no way be considered definitive nor comprehensive. And yes, there are other things besides performance that are important when considering a dbm.

1.3.1 Benchmarking Approach

The benchmarks used for semidbm are based off the benchmark scripts for leveldb. You can run the benchmark scripts yourself using the *scripts/benchmark* script in the repo. By default, the benchmark uses a db of one million keys with 16 byte keys and 100 byte values (these are the values used for leveldb's benchmarks). All of these parameters can be changed via command line arugments (*-n*, *-k*, *-s* respectively).

The benchmark script is written in a way to be compatible with any module supporting the dbm interface. Given the dbm interface isn't entirely standardized, this is what is required:

- An *open()* function in the module (that behaves like dumbdbm.open, gdbm.open, etc).
- The returned object from *open()* is a "dbm" like object. All the object needs to support is *__getitem__*, *__setitem__*, *__delitem__*, and *close()*.

To specify what dbm module to use, use the *-d* argument. The value of this argument should the module name of the dbm, for example, to run the benchmarks against semidbm:

```
scripts/benchmark -d semidbm
```

The *-d* argument can be specified multiple times.

If a dbm does not support a dbm interface, an adapter module can be written for the dbm. The directory *scripts/adapters* is added to *sys.path* before the benchmarks are run, so benchmarking a 3rd party dbm is straightforward. For example, in order to benchmark Berkeley DB using the bsddb3 module, a *scripts/adapters/bdb_minimal.py* file was created:

```
import bsddb3.db
def open(filename, mode):
    db = bsddb3.db.DB(None)
    if mode == 'r':
        flags = bsddb3.db.DB_RDONLY
    elif mode == 'rw':
        flags = 0
    elif mode == 'w':
        flags =  bsddb3.db.DB_CREATE
    elif mode == 'c':
        flags =  bsddb3.db.DB_CREATE
    elif mode == 'n':
        flags = bsddb3.db.DB_TRUNCATE | bsddb3.db.DB_CREATE
    else:
```

```
        raise bsddb3.db.DBError(
            "flags should be one of 'r', 'w', 'c' or 'n' or use the "
            "bsddb.db.DB_* flags")
    db.open(filename, None, bsddb3.db.DB_HASH, flags)
    return db
```

The bsddb3.db.DB object can now be benchmarked using:

```
scripts/benchmark -d bdb_minimal
```

1.3.2 Benchmark Results

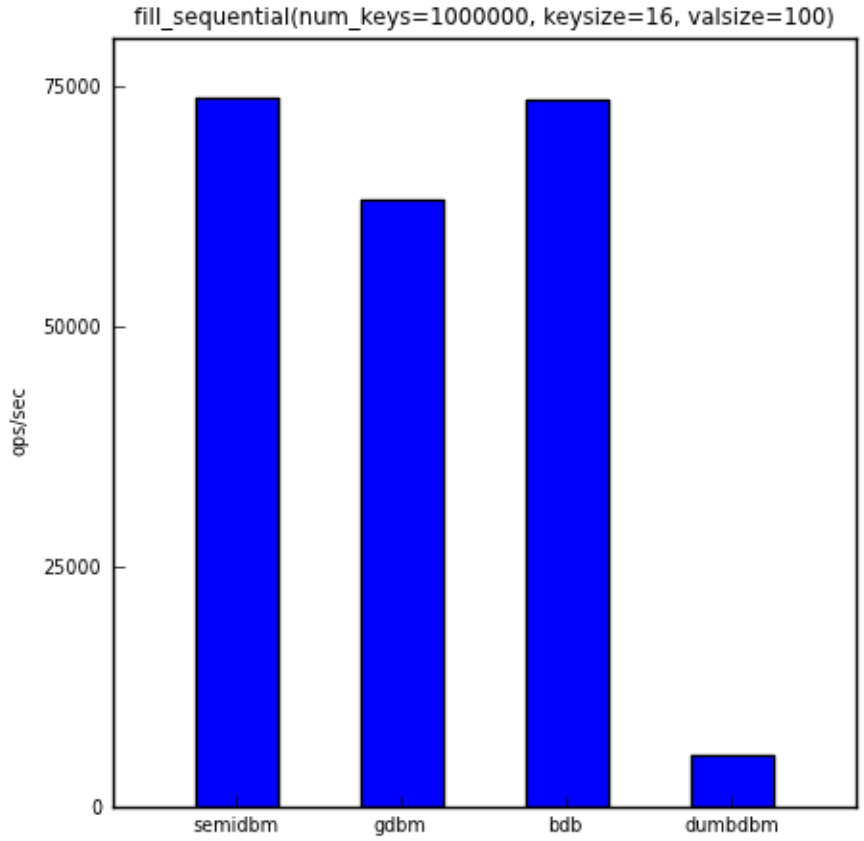
Below are the results of benchmarking various dbms. Although *scripts/benchmark* shows the results in various forms of measurement, the measurement chosen here is the average number of operations per second over the total number of keys. For this measurement, **higher is better**.

The dbms chosen for this benchmark are:

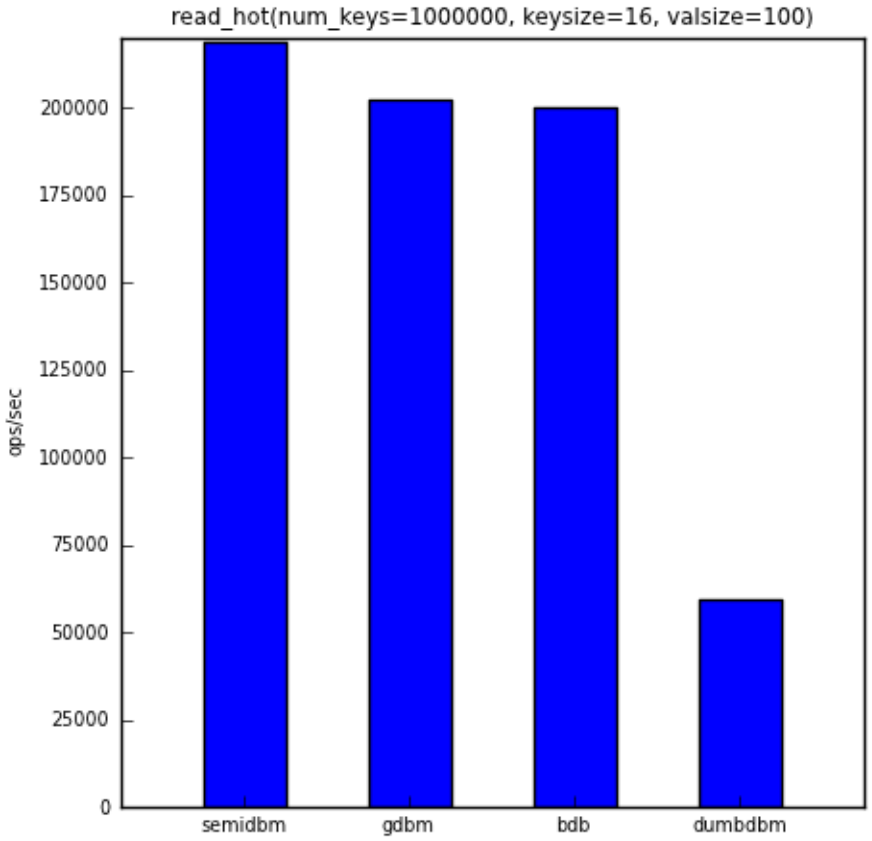
- semidbm
- gdbm (GDN dbm)
- bdb (minimal Berkeley DB interface, *scripts/adapaters/bdb_minimal.py*)
- dumbdbm

The *dbm* module was not included because it was not able to add 1000000 to its db, it raises an exception around 420000 keys with an "Out of overflow pages" error.

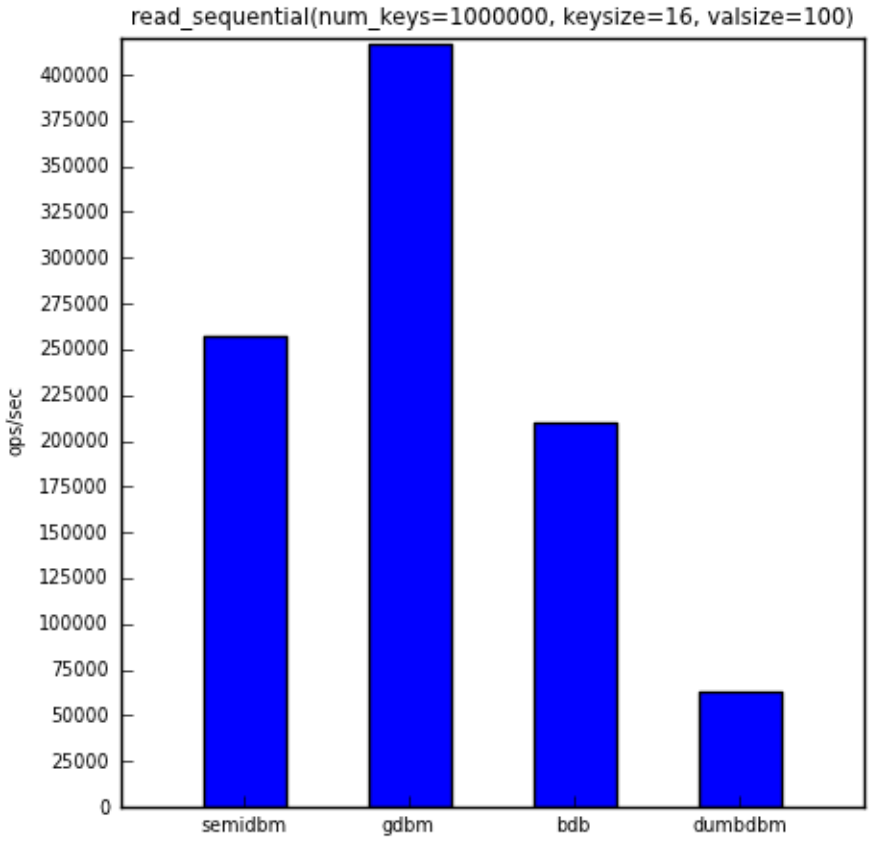
This first benchmark shows the ops/sec for adding one million keys to the db.



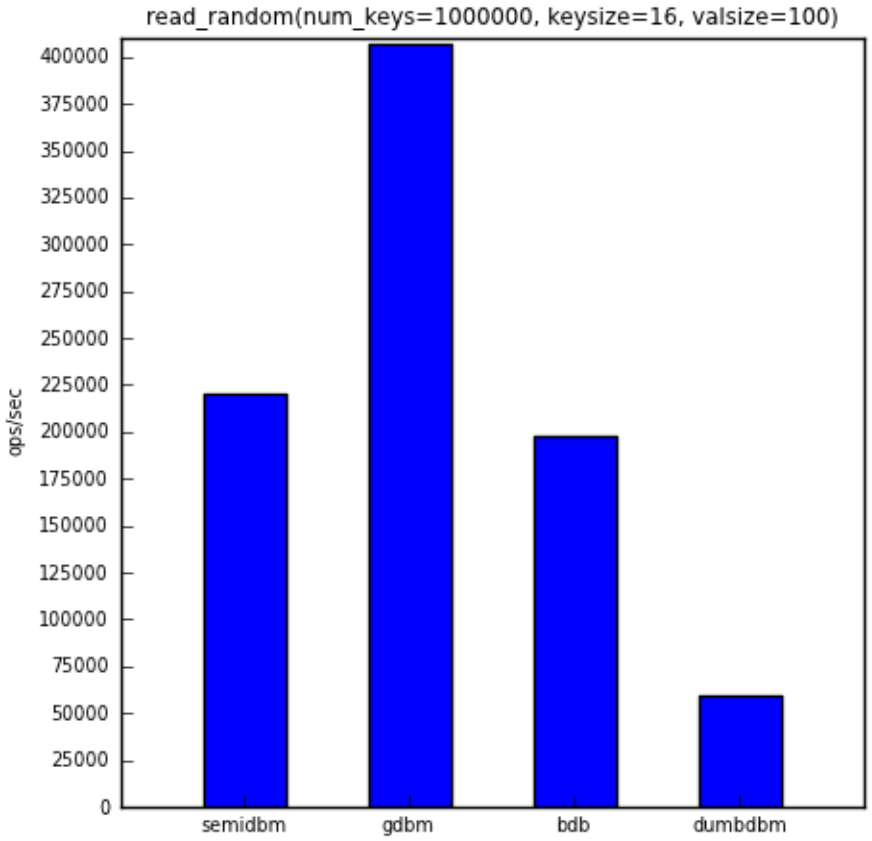
The second benchmark shows the ops/sec for repeatedly accessing 1% of the keys (randomly selected).



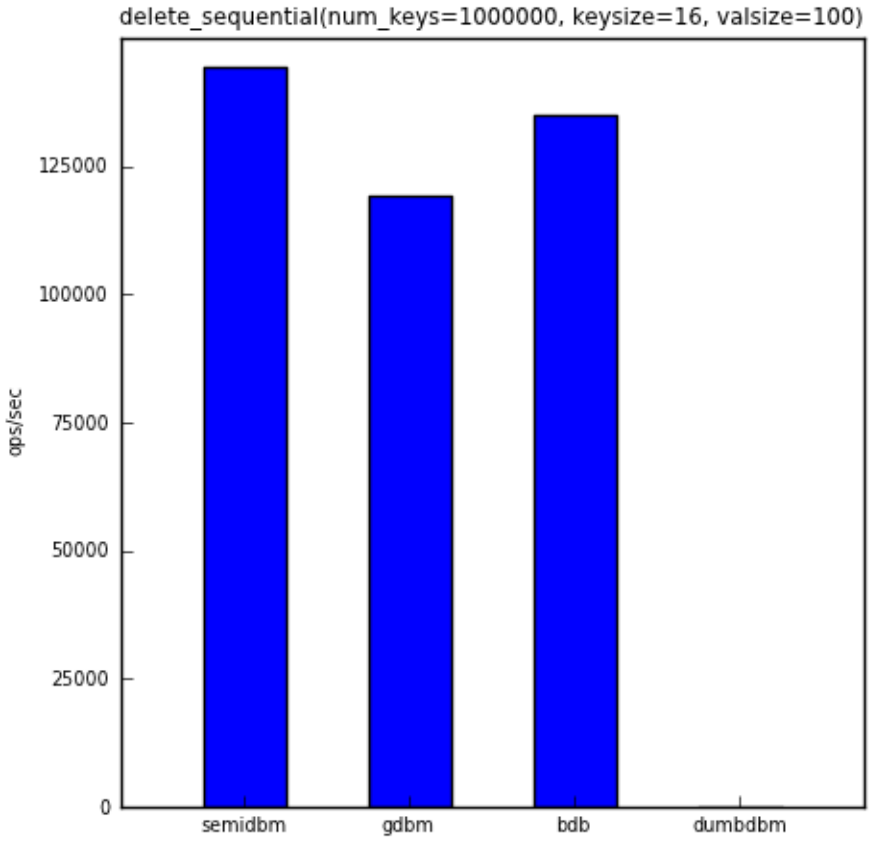
The next benchmark shows the ops/sec for reading all one million keys in the same order that they were added.



The next benchmark shows the ops/sec for reading all one million keys in a randomly selected order.



And the last benchmark shows the ops/sec for deleting all one million keys in the same order that they were added.



Note that dumbdbm is not shown in the chart above. This is because deletion of keys in dumbdbm is extremely slow. It also appears to have O(n) behavior (it writes out its data file on every delete). To give you an idea of the performance, running this benchmark against dumbdbm with 1000 keys gave an average ops/sec for the delete_sequential benchmark of **800**. For 10000 keys dumbdbm resulted in **104** ops/sec.

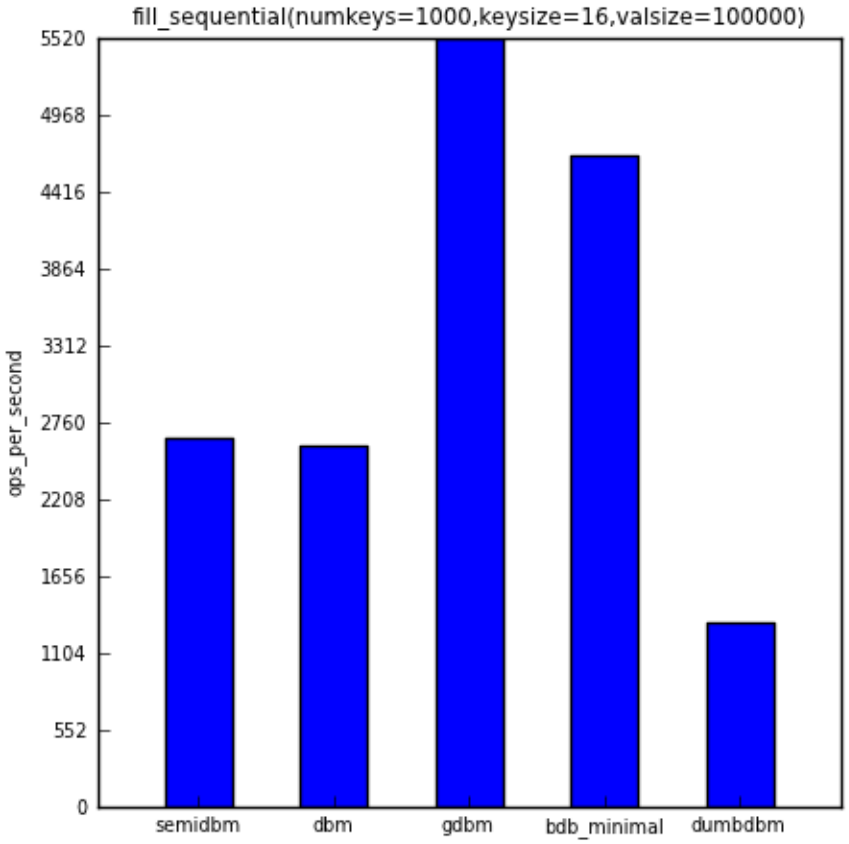
The table below shows the actual numbers for the charts above.

	semidbm	**gdbm**	**bdb**	**dumbdbm**
fill_sequential	**73810**	63177	73614	5460
read_hot	**218651**	202432	200111	59569
read_sequential	257668	**417320**	209696	62605
read_random	219962	**406594**	197690	59258
delete_sequential	**144265**	119167	135137	0

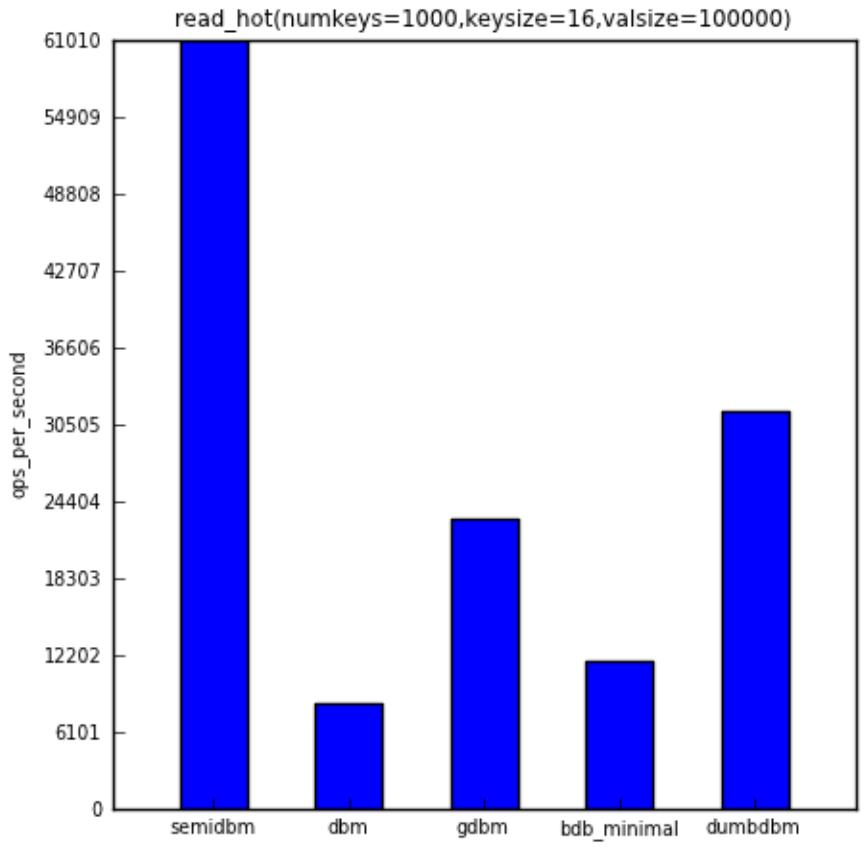
Benchmarking With Large Values

One area where semidbm benchmarks really well is when dealing with large values. The same 5 benchmarks were repeated, but with only 1000 total keys, 16 byte keys, and 100000 byte values.

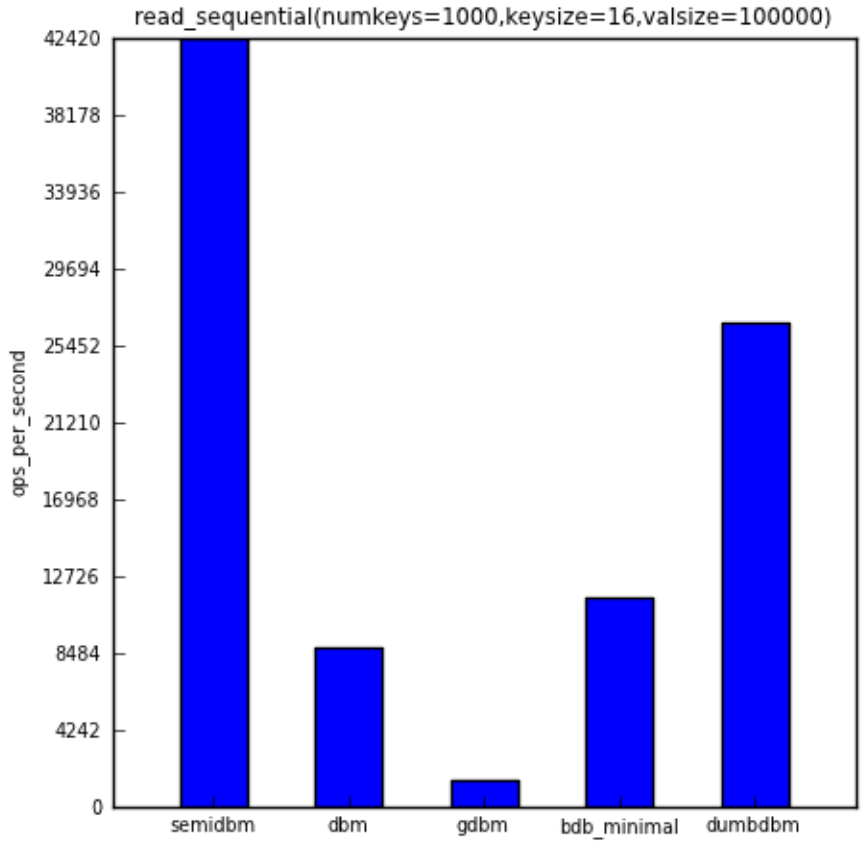
The first benchmark shows the ops/sec for 1000 sequential writes.



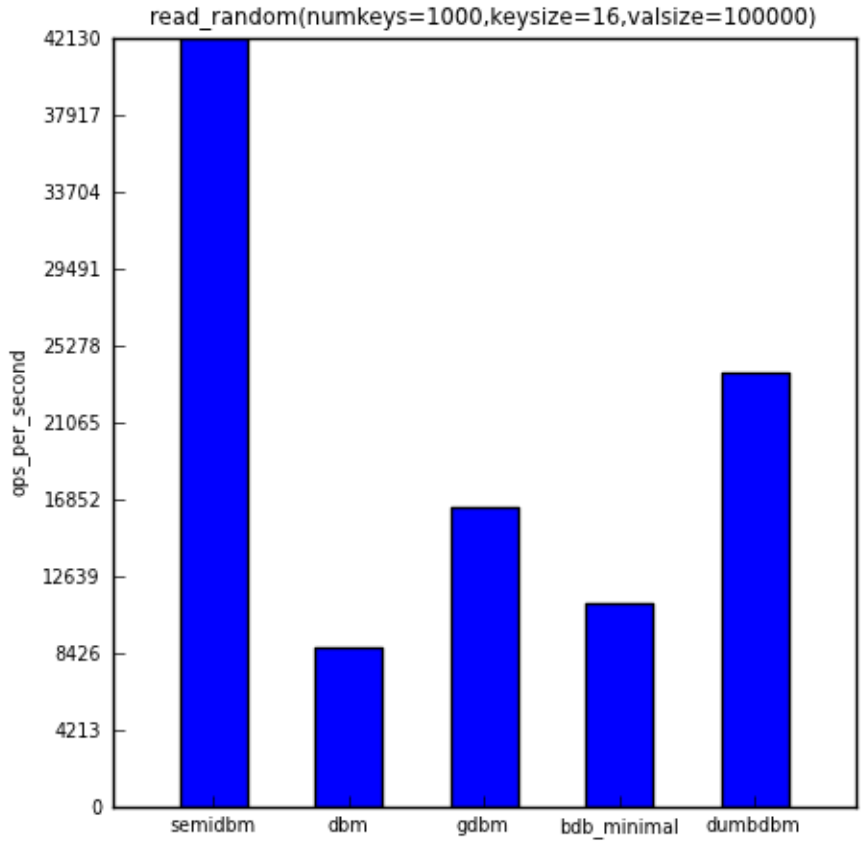
The second benchmark shows the ops/sec for repeatedly accessing 1% of the keys (randomly selected).



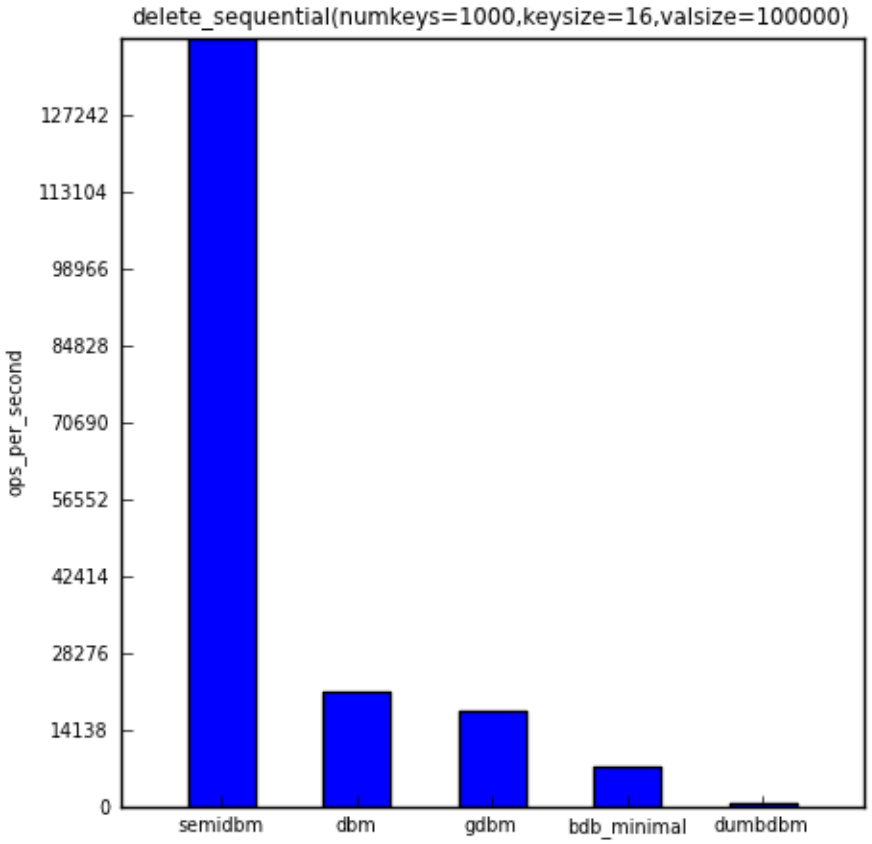
The third benchmark shows the ops/sec for sequentially reading all 1000 keys.



The fourth benchmark shows the ops/sec for reading all 1000 keys in a randomly selected order.



And the last benchmark shows the ops/sec for deleting all 1000 keys in the same order that they were added.



Below is the raw data used to generate the above charts.

n=1000,k=16,v=100000	semidbm	dbm	gdbm	bdb_minimal	dumbdbm
fill_sequential	2653	2591	**5525**	4677	1330
read_hot	**61016**	8363	23104	11782	31624
read_sequential	**42421**	8822	1508	11519	26757
read_random	**42133**	8720	16442	11162	23778
delete_sequential	**141379**	21167	17695	7267	780

You can see that with the exception of fill_sequential (in which the fastest module, gdbm, was roughly twice as fast as semidbm), semidbm completely outperforms all the other dbms. In the case of read_sequential, semidbm's **28 times faster than gdbm.**

Overall, semidbm's performance is comparable to the performance of other dbms with small keys and values, but is surprisingly faster than other dbms when reading large values. It's also clear that semidbm is faster than dumbdbm is all of the benchmarks shown here.

Running the Benchmarks

You are encouraged to run the benchmarks yourself, to recreate the benchmark above, you can run:

```
scripts/benchmark -d semidbm -d gdbm -d bdb_minimal -d dumbdbm
```

Though keep in mind that you will probably want to stop the benchmark once dumbdbm reaches the delete_sequential benchmark. Either that or you can leave off dumbdbm and run it with a smaller number of keys:

```
scripts/benchmark -d dumbdbm -n 10000
```

1.4 Changelog

1.4.1 0.5.0

- Remove mmap read only dbm subclass. This functionality has not been available in a public interface since b265e60c5f4c0b1e8e9e4343f5f2300b5e017bf0 (1.5 years ago) so it's now removed.
- Added non mmap based dbm loader for platforms that do not support mmap.
- Atomic renames on windows during db compaction.

1.4.2 0.4.0

0.4.0 is a backwards incompatible release with 0.3.1. Data files created with 0.3.1 will not work with 0.4.0. The reasons for switching to 0.4.0 include:

- Data format switched from ASCII to binary file format, this resulted in a nice performance boost.
- Index and data file consolidated to a single file, resulting in improved write performance.
- Checksums are written for all entries. Checksums can be verified for every __getitem__ call (off by default).
- Python 3 support (officially python 3.3.x).

1.4.3 0.3.1

- Windows support.

1.4.4 0.3.0

- The data file and the index file are kept in a separate directory. To load the the db you specify the directory name instead of the data filename.
- Non-mmapped read only version is used when the db is opened with `r`.
- Write performance improvements.

1.4.5 0.2.1

- DB can be opened with `r`, `c`, `w`, and `n`.
- Add a memory mapped read only implementation for reading from the DB (if your entire data file can be mmapped this provides a huge performance boost for reads).
- Benchmark scripts rewritten to provide more useful information.

1.4.6 0.2.0

- New `sync()` method to ensure data is written to disk.
  - `sync()` is called during compaction and on `close()`.
- Add a `DBMLoadError` exception for catching semidbm loading errors.

CHAPTER

TWO

DEVELOPER DOCUMENTATION

2.1 API for semidbm

semidbm.db.**open**(*filename*, *flag='r'*, *mode=438*, *verify_checksums=False*)
Open a semidbm database.

Parameters

- **filename** – The name of the db. Note that for semidbm, this is actually a directory name. The argument is named *filename* to be compatible with the dbm interface.
- **flag** – Specifies how the db should be opened. *flag* can be any of these values:

Value	Meaning
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

- **mode** – Not currently used (provided to be compatible with the dbm interface).
- **verify_checksums** – Verify the checksums for each value are correct on every __getitem__ call (defaults to False).

class semidbm.db.**_SemiDBM**(*dbdir*, *renamer*, *data_loader=None*, *verify_checksums=False*)

Parameters **dbdir** – The directory containing the dbm files. If the directory does not exist it will be created.

close(*compact=False*)
Close the db.

The data is synced to disk and the db is closed. Once the db has been closed, no further reads or writes are allowed.

Parameters **compact** – Indicate whether or not to compact the db before closing the db.

compact()
Compact the db to reduce space.

This method will compact the data file and the index file. This is needed because of the append only nature of the index and data files. This method will read the index and data file and write out smaller but equivalent versions of these files.

As a general rule of thumb, the more non read updates you do, the more space you'll save when you compact.

keys()
Return all they keys in the db.

The keys are returned in an arbitrary order.

sync()
Sync the db to disk.

This will flush any of the existing buffers and fsync the data to disk.

You should call this method to guarantee that the data is written to disk. This method is also called whenever the dbm is *close()*'d.

2.2 File Format of DB file

author James Saryerwinnie

status Draft

target-version 0.4.0

date April 15, 2013

2.2.1 Abstract

This document proposes a new file format for semidbm. This is a backwards incompatible change.

2.2.2 Motivation

When python3 support was added, `semidbm` received a significant performance degredation. This was mainly due to the str vs. bytes differentiation, and the fact that semidbm was a text based format. All of the integer sizes and checksum information was written as ASCII strings, and as a result, encoding the string to a byte sequence added additional overhead.

In order to improve performance, `semidbm` should adopt a binary format, specifically the sizes of the keys and values as well as the checksums should be written as binary values. This will avoid the need to use string formatting when writing values. It will also improve the load time of a db file.

2.2.3 Specification

A semidbm file will consist of a header and a sequence of entries. All multibyte sequences are writteni network byte order.

2.2.4 Header

The semidbm header format consists of:

- 4 byte magic number (`53 45 4d 49`)
- 4 byte version number consisting of 2 byte major version and 2 byte minor version (currently (1, 1)).

2.2.5 Entries

After the header, the file contains a sequence of entries. Each entry has this format:

- 4 byte key size
- 4 byte value size
- Key contents
- Value content
- 4 byte CRC32 checksum of Key + Value

If a key is deleted it will have a value size of -1 and no value content.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*