Plyvel

Release 1.3.0

Dec 15, 2020

Contents

Plyvel is a fast and feature-rich Python interface to LevelDB.

Plyvel's key features are:

- **Rich feature set**

  Plyvel wraps most of the LevelDB C++ API and adds some features of its own. In addition to basic features like getting, putting and deleting data, Plyvel allows you to use write batches, database snapshots, very flexible iterators, prefixed databases, bloom filters, custom cache sizes, custom comparators, and other goodness LevelDB has to offer.

- **Friendly Pythonic API**

  Plyvel has a friendly and well-designed API that uses Python idioms like iterators and context managers (`with` blocks), without sacrificing the power or performance of the underlying LevelDB C++ API.

- **High performance**

  Plyvel executes all performance-critical code at C speed (using Cython), which means Plyvel is a good fit for high performance applications.

- **Extensive documentation**

  Plyvel comes with extensive documentation, including a user guide and API reference material.

You should know that Plyvel is a hobby project, written and maintained by me, Wouter Bolsterlee, in my spare time. Please consider making a small donation to let me know you appreciate my work. Thanks!

Documentation contents

CHAPTER 1

Installation guide

This guide provides installation instructions for Plyvel.

1.1 Build and install Plyvel

The recommended (and easiest) way to install Plyvel is to install it into a virtual environment (*virtualenv*):

```
$ virtualenv envname
$ source envname/bin/activate
```

Now you can automatically install the latest Plyvel release from the Python Package Index (PyPI) using `pip`:

```
(envname) $ pip install plyvel
```

(In case you're feeling old-fashioned: downloading a source tarball, unpacking it and installing it manually with `python setup.py install` should also work.)

The Plyvel source package does not include a copy of LevelDB itself. Plyvel requires LevelDB development headers and an installed shared library for LevelDB during build time, and the same installed shared library at runtime.

To build from source, make sure you have a shared LevelDB library and the development headers installed where the compiler and linker can find them. For Debian or Ubuntu something like `apt-get install libleveldb1v5 libleveldb-dev` should suffice.

For Linux, Plyvel also ships as pre-built binary packages (`manylinux1` wheels) that have LevelDB embedded. Simply running `pip install plyvel` does the right thing with a modern `pip` on a modern Linux platform, even without any LevelDB libraries on your system.

Note: Plyvel 1.x depends on LevelDB >= 1.20, which at the time of writing (early 2018) is more recent than the versions packaged by various Linux distributions. Using an older version will result in compile-time errors. The easiest solution is to use the pre-built binary packages. Alternatively, install LevelDB manually on your system. The

Dockerfile in the Plyvel source repository, which is used for building the official binary packages, shows how to do this.

Warning: The above installation method applies only to released packages available from PyPI. If you are building and installing from a source tree acquired through other means, e.g. checked out from source control, you will need to run Cython first. If you don't, you will see errors about missing source files. See the *developer documentation* for more information.

1.2 Verify that it works

After installation, this command should not give any output:

```
(envname) $ python -c 'import plyvel'
```

If you see an `ImportError` complaining about undefined symbols, e.g.

```
ImportError: ./plyvel.so: undefined symbol: _ZN7leveldb10WriteBatch5ClearEv
```

...then the installer (actually, the linker) was unable to find the LevelDB library on your system when building Plyvel. Install LevelDB or set the proper environment variables for the compiler and linker and try `pip install --reinstall plyvel`.

Next steps

Continue with the *user guide* to see how to use Plyvel.

CHAPTER 2

User guide

This user guide gives an overview of Plyvel. It covers:

- opening and closing databases,
- storing and retrieving data,
- working with write batches,
- using snapshots,
- iterating over your data,
- using prefixed databases, and
- implementing custom comparators.

Note: this document assumes basic familiarity with LevelDB; visit the LevelDB homepage for more information about its features and design.

2.1 Getting started

After *installing Plyvel*, we can simply import `plyvel`:

```
>>> import plyvel
```

Let's open a new database by creating a new *DB* instance:

```
>>> db = plyvel.DB('/tmp/testdb/', create_if_missing=True)
```

That's all there is to it. At this point `/tmp/testdb/` contains a fresh LevelDB database (assuming the directory did not contain a LevelDB database already).

For real world applications, you probably want to tweak things like the size of the memory cache and the number of bits to use for the (optional) bloom filter. These settings, and many others, can be specified as arguments to the *DB* constructor. For this tutorial we'll just use the LevelDB defaults.

To close the database we just opened, use *DB.close()* and inspect the `closed` property:

```
>>> db.closed
False
>>> db.close()
>>> db.closed
True
```

Alternatively, you can just delete the variable that points to it, but this might not close the database immediately, e.g. because active iterators are using it:

```
>>> del db
```

Note that the remainder of this tutorial assumes an open database, so you probably want to skip the above if you're performing all the steps in this tutorial yourself.

2.2 Basic operations

Now that we have our database, we can use the basic LevelDB operations: putting, getting, and deleting data. Let's look at these in turn.

First we'll add some data to the database by calling *DB.put()* with a key/value pair:

```
>>> db.put(b'key', b'value')
>>> db.put(b'another-key', b'another-value')
```

To get the data out again, use *DB.get()*:

```
>>> db.get(b'key')
'value'
```

If you try to retrieve a key that does not exist, a `None` value is returned:

```
>>> print(db.get(b'yet-another-key'))
None
```

Optionally, you can specify a default value, just like `dict.get()`:

```
>>> print(db.get(b'yet-another-key', b'default-value'))
'default-value'
```

Finally, to delete data from the database, use *DB.delete()*:

```
>>> db.delete(b'key')
>>> db.delete(b'another-key')
```

At this point our database is empty again. Note that, in addition to the basic use shown above, the *put()*, *get()*, and *delete()* methods accept optional keyword arguments that influence their behaviour, e.g. for synchronous writes or reads that will not fill the cache.

2.3 Write batches

LevelDB provides *write batches* for bulk data modification. Since batches are faster than repeatedly calling *DB.put()* or *DB.delete()*, batches are perfect for bulk loading data. Let's write some data:

```
>>> wb = db.write_batch()
>>> for i in xrange(100000):
        wb.put(str(i).encode(), str(i).encode() * 100)
...
>>> wb.write()
```

Since write batches are committed in an atomic way, either the complete batch is written, or not at all, so if your machine crashes while LevelDB writes the batch to disk, the database will not end up containing partial or inconsistent data. This makes write batches very useful for multiple modifications to the database that should be applied as a group.

Write batches can also act as context managers. The following code does the same as the example above, but there is no call to *WriteBatch.write()* anymore:

```
>>> with db.write_batch() as wb:
...     for i in xrange(100000):
...         wb.put(str(i).encode(), str(i).encode() * 100)
```

If the `with` block raises an exception, pending modifications in the write batch will still be written to the database. This means each modification using *put()* or *delete()* that happened before the exception was raised will be applied to the database:

```
>>> with db.write_batch() as wb:
...     wb.put(b'key-1', b'value-1')
...     raise ValueError("Something went wrong!")
...     wb.put(b'key-2', b'value-2')
```

At this point the database contains `key-1`, but not `key-2`. Sometimes this behaviour is undesirable. If you want to discard all pending modifications in the write batch if an exception occurs, you can simply set the *transaction* argument:

```
>>> with db.write_batch(transaction=True) as wb:
...     wb.put(b'key-3', b'value-3')
...     raise ValueError("Something went wrong!")
...     wb.put(b'key-4', b'value-4')
```

In this case the database will not be modified, because the `with` block raised an exception. In this example this means that neither `key-3` nor `key-4` will be saved.

Note: Write batches will never silently suppress exceptions. Exceptions will be propagated regardless of the value of the *transaction* argument, so in the examples above you will still see the ValueError.

2.4 Snapshots

A snapshot is a consistent read-only view over the entire database. Any data that is modified after the snapshot was taken, will not be seen by the snapshot. Let's store a value:

```
>>> db.put(b'key', b'first-value')
```

Now we'll make a snapshot using *DB.snapshot()*:

```
>>> sn = db.snapshot()
>>> sn.get(b'key')
'first-value'
```

At this point any modifications to the database will not be visible by the snapshot:

```
>>> db.put(b'key', b'second-value')
>>> sn.get(b'key')
'first-value'
```

Long-lived snapshots may consume significant resources in your LevelDB database, since the snapshot prevents LevelDB from cleaning up old data that is still accessible by the snapshot. This means that you should never keep a snapshot around longer than necessary. The snapshot and its associated resources will be released automatically when the snapshot reference count drops to zero, which (for local variables) happens when the variable goes out of scope (or after you've issued a `del` statement). If you want explicit control over the lifetime of a snapshot, you can also clean it up yourself using *`Snapshot.close()`*:

```
>>> sn.close()
```

Alternatively, you can use the snapshot as a context manager:

```
>>> with db.snapshot() as sn:
...     sn.get(b'key')
```

2.5 Iterators

All key/value pairs in a LevelDB database will be sorted by key. Because of this, data can be efficiently retrieved in sorted order. This is what iterators are for. Iterators allow you to efficiently iterate over all sorted key/value pairs in the database, or more likely, a range of the database.

Let's fill the database with some data first:

```
>>> db.put(b'key-1', b'value-1')
>>> db.put(b'key-5', b'value-5')
>>> db.put(b'key-3', b'value-3')
>>> db.put(b'key-2', b'value-2')
>>> db.put(b'key-4', b'value-4')
```

Now we can iterate over all data using a simple `for` loop, which will return all key/value pairs in lexicographical key order:

```
>>> for key, value in db:
...     print(key)
...     print(value)
...
key-1
value-1
key-2
value-2
key-3
value-3
key-4
value-4
key-5
```

While the complete database can be iterated over by just looping over the *`DB`* instance, this is generally not useful. The *`DB.iterator()`* method allows you to obtain more specific iterators. This method takes several optional arguments to specify how the iterator should behave.

2.5.1 Iterating over a key range

Limiting the range of values that you want the iterator to iterate over can be achieved by supplying *start* and/or *stop* arguments:

```
>>> for key, value in db.iterator(start=b'key-2', stop=b'key-4'):
...     print(key)
...
key-2
key-3
```

Any combination of *start* and *stop* arguments is possible. For example, to iterate from a specific start key until the end of the database:

```
>>> for key, value in db.iterator(start=b'key-3'):
...     print(key)
...
key-3
key-4
key-5
```

By default the start key is *inclusive* and the stop key is *exclusive*. This matches the behaviour of Python's built-in `range()` function. If you want different behaviour, you can use the *include_start* and *include_stop* arguments:

```
>>> for key, value in db.iterator(start=b'key-2', include_start=False,
...                               stop=b'key-5', include_stop=True):
...     print(key)
key-3
key-4
key-5
```

Instead of specifying *start* and *stop* keys, you can also specify a *prefix* for keys. In this case the iterator will only return key/value pairs whose key starts with the specified prefix. In our example, all keys have the same prefix, so this will return all key/value pairs:

```
>>> for key, value in db.iterator(prefix=b'ke'):
...     print(key)
key-1
key-2
key-3
key-4
key-5
>>> for key, value in db.iterator(prefix=b'key-4'):
...     print(key)
key-4
```

2.5.2 Limiting the returned data

If you're only interested in either the key or the value, you can use the *include_key* and *include_value* arguments to omit data you don't need:

```
>>> list(db.iterator(start=b'key-2', stop=b'key-4', include_value=False))
['key-2', 'key-3']
>>> list(db.iterator(start=b'key-2', stop=b'key-4', include_key=False))
['value-2', 'value-3']
```

Only requesting the data that you are interested in results in slightly faster iterators, since Plyvel will avoid unnecessary memory copies and object construction in this case.

2.5.3 Iterating in reverse order

LevelDB also supports reverse iteration. Just set the *reverse* argument to *True* to obtain a reverse iterator:

```
>>> list(db.iterator(start=b'key-2', stop=b'key-4', include_value=False,␣
↪reverse=True))
['key-3', 'key-2']
```

Note that the *start* and *stop* keys are the same; the only difference is the *reverse* argument.

2.5.4 Iterating over snapshots

In addition to directly iterating over the database, LevelDB also supports iterating over snapshots using the *Snapshot.iterator()* method. This method works exactly the same as *DB.iterator()*, except that it operates on the snapshot instead of the complete database.

2.5.5 Closing iterators

It is generally not required to close an iterator explicitly, since it will be closed when it goes out of scope (or is garbage collected). However, due to the way LevelDB is designed, each iterator operates on an implicit database snapshot, which can be an expensive resource depending on how the database is used during the iterator's lifetime. The *Iterator.close()* method gives explicit control over when those resources are released:

```
>>> it = db.iterator()
>>> it.close()
```

Alternatively, to ensure that an iterator is immediately closed after use, you can also use it as a context manager using the `with` statement:

```
>>> with db.iterator() as it:
...     for k, v in it:
...         pass
```

2.5.6 Non-linear iteration

In the examples above, we've only used Python's standard iteration methods using a `for` loop and the `list()` constructor. This suffices for most applications, but sometimes more advanced iterator tricks can be useful. Plyvel exposes pretty much all features of the LevelDB iterators using extra functions on the *Iterator* instance that *DB.iterator()* and *Snapshot.iterator()* returns.

For instance, you can step forward and backward over the same iterator. For forward stepping, Python's standard `next()` built-in function can be used (this is also what a standard `for` loop does). For backward stepping, you will need to call the *prev()* method on the iterator:

```
>>> it = db.iterator(include_value=False)
>>> next(it)
'key-1'
>>> next(it)
'key-2'
```

(continues on next page)

(continued from previous page)

```
>>> next(it)
'key-3'
>>> it.prev()
'key-3'
>>> next(it)
'key-3'
>>> next(it)
'key-4'
>>> next(it)
'key-5'
>>> next(it)
Traceback (most recent call last):
  ...
StopIteration

>>> it.prev()
'key-5'
```

Note that for reverse iterators, the definition of 'forward' and 'backward' is inverted, i.e. calling `next(it)` on a reverse iterator will return the key that sorts *before* the key that was most recently returned.

Additionally, Plyvel supports seeking on iterators:

```
>>> it = db.iterator(include_value=False)
>>> it.seek(b'key-3')
>>> next(it)
'key-3'
>>> it.seek_to_start()
>>> next(it)
'key-1'
```

See the *Iterator* API reference for more information about advanced iterator usage.

2.5.7 Raw iterators

In addition to the iterators describe above, which adhere to the Python iterator protocol, there is also a *raw iterator* API that mimics the C++ iterator API provided by LevelDB. Since this interface is only intended for advanced use cases, it is not covered in this user guide. See the API reference for *DB.raw_iterator()* and *RawIterator* for more information.

2.6 Prefixed databases

LevelDB databases have a single key space. A common way to split a LevelDB database into separate partitions is to use a prefix for each partition. Plyvel makes this very easy to do using the *DB.prefixed_db()* method:

```
>>> my_sub_db = db.prefixed_db(b'example-')
```

The `my_sub_db` variable in this example points to an instance of the *PrefixedDB* class. This class behaves mostly like a normal Plyvel *DB* instance, but all operations will transparently add the key prefix to all keys that it accepts (e.g. in *PrefixedDB.get()*), and strip the key prefix from all keys that it returns (e.g. from *PrefixedDB.iterator()*). Examples:

```
>>> my_sub_db.get(b'some-key')  # this looks up b'example-some-key'
>>> my_sub_db.put(b'some-key', b'value')  # this sets b'example-some-key'
```

Almost all functionality available on *DB* is also available from *PrefixedDB*: write batches, iterators, snapshots, and also iterators over snapshots. A *PrefixedDB* is simply a lightweight object that delegates to the the real *DB*, which is accessible using the *db* attribute:

```
>>> real_db = my_sub_db.db
```

You can even nest key spaces by creating prefixed prefixed databases using *PrefixedDB.prefixed_db()*:

```
>>> my_sub_sub_db = my_sub_db.prefixed_db(b'other-prefix')
```

2.7 Custom comparators

LevelDB provides an ordered data store, which means all keys are stored in sorted order. By default, a byte-wise comparator that works like `strcmp()` is used, but this behaviour can be changed by providing a custom comparator. Plyvel allows you to use a Python callable as a custom LevelDB comparator.

The signature for a comparator callable is simple: it takes two byte strings and should return either a positive number, zero, or a negative number, depending on whether the first byte string is greater than, equal to or less than the second byte string. (These are the same semantics as the built-in `cmp()`, which has been removed in Python 3 in favour of the so-called 'rich' comparison methods.)

A simple comparator function for case insensitive comparisons might look like this:

```
def comparator(a, b):
    a = a.lower()
    b = b.lower()

    if a < b:
        # a sorts before b
        return -1

    if a > b:
        # a sorts after b
        return 1

    # a and b are equal
    return 0
```

(This is a toy example. It only works properly for byte strings with characters in the ASCII range.)

To use this comparator, pass the *comparator* and *comparator_name* arguments to the *DB* constructor:

```
>>> db = DB('/path/to/database/',
...         comparator=comparator,  # the function defined above
...         comparator_name=b'CaseInsensitiveComparator')
```

The comparator name, which must be a byte string, will be stored in the database. LevelDB refuses to open existing databases if the provided comparator name does not match the one in the database.

LevelDB invokes the comparator callable repeatedly during many of its operations, including storing and retrieving data, but also during background compactions. Background compaction uses threads that are 'invisible' from Python. This means that custom comparator callables *must not* raise any exceptions, since there is no proper way to recover

from those. If an exception happens nonetheless, Plyvel will print the traceback to *stderr* and immediately abort your program to avoid database corruption.

A final thing to keep in mind is that custom comparators written in Python come with a considerable performance impact. Experiments with simple Python comparator functions like the example above show a 4× slowdown for bulk writes compared to the built-in LevelDB comparator.

Next steps

The user guide should be enough to get you started with Plyvel. A complete description of the Plyvel API is available from the *API reference*.

CHAPTER 3

API reference

This document is the API reference for Plyvel. It describes all classes, methods, functions, and attributes that are part of the public API.

Most of the terminology in the Plyvel API comes straight from the LevelDB API. See the LevelDB documentation and the LevelDB header files (`$prefix/include/leveldb/*.h`) for more detailed explanations of all flags and options.

3.1 Database

Plyvel exposes the *DB* class as the main interface to LevelDB. Application code should create a *DB* and use the appropriate methods on this instance to create write batches, snapshots, and iterators for that LevelDB database.

class DB

LevelDB database

__init__(*name*, *create_if_missing=False*, *error_if_exists=False*, *paranoid_checks=None*, *write_buffer_size=None*, *max_open_files=None*, *lru_cache_size=None*, *block_size=None*, *block_restart_interval=None*, *max_file_size=None*, *compression='snappy'*, *bloom_filter_bits=0*, *comparator=None*, *comparator_name=None*)

Open the underlying database handle.

Most arguments have the same name as the the corresponding LevelDB parameters; see the LevelDB documentation for a detailed description. Arguments defaulting to *None* are only propagated to LevelDB if specified, e.g. not specifying a *write_buffer_size* means the LevelDB defaults are used.

Most arguments are optional; only the database name is required.

See the descriptions for `DB`, `DB::Open()`, `Cache`, `FilterPolicy`, and `Comparator` in the LevelDB C++ API for more information.

New in version 1.0.0: *max_file_size* argument

Parameters

- **name** (*str*) – name of the database (directory name)

- **create_if_missing** (*bool*) – whether a new database should be created if needed
- **error_if_exists** (*bool*) – whether to raise an exception if the database already exists
- **paranoid_checks** (*bool*) – whether to enable paranoid checks
- **write_buffer_size** (*int*) – size of the write buffer (in bytes)
- **max_open_files** (*int*) – maximum number of files to keep open
- **lru_cache_size** (*int*) – size of the LRU cache (in bytes)
- **block_size** (*int*) – block size (in bytes)
- **block_restart_interval** (*int*) – block restart interval for delta encoding of keys
- **max_file_size** (*bool*) – maximum file size (in bytes)
- **compression** (*bool*) – whether to use Snappy compression (enabled by default))
- **bloom_filter_bits** (*int*) – the number of bits to use for a bloom filter; the default of 0 means that no bloom filter will be used
- **comparator** (*callable*) – a custom comparator callable that takes two byte strings and returns an integer
- **comparator_name** (*bytes*) – name for the custom comparator

name
The (directory) name of this *DB* instance. This is a *read-only* attribute and must be set at instantiation time.

New in version 1.1.0.

close()
Close the database.

This closes the database and releases associated resources such as open file pointers and caches.

Any further operations on the closed database will raise `RuntimeError`.

Warning: Closing a database while other threads are busy accessing the same database may result in hard crashes, since database operations do not perform any synchronisation/locking on the database object (for performance reasons) and simply assume it is available (and open). Applications should make sure not to close databases that are concurrently used from other threads.

See the description for `DB` in the LevelDB C++ API for more information. This method deletes the underlying DB handle in the LevelDB C++ API and also frees other related objects.

closed
Boolean attribute indicating whether the database is closed.

get(*key*, *default=None*, *verify_checksums=False*, *fill_cache=True*)
Get the value for the specified key, or *default* if no value was set.

See the description for `DB::Get()` in the LevelDB C++ API for more information.

New in version 0.4: *default* argument

Parameters

- **key** (*bytes*) – key to retrieve
- **default** – default value if key is not found

- **verify_checksums** (*bool*) – whether to verify checksums
- **fill_cache** (*bool*) – whether to fill the cache

Returns value for the specified key, or *None* if not found

Return type bytes

put(*key*, *value*, *sync=False*)
Set a value for the specified key.

See the description for `DB::Put()` in the LevelDB C++ API for more information.

Parameters

- **key** (*bytes*) – key to set
- **value** (*bytes*) – value to set
- **sync** (*bool*) – whether to use synchronous writes

delete(*key*, *sync=False*)
Delete the key/value pair for the specified key.

See the description for `DB::Delete()` in the LevelDB C++ API for more information.

Parameters

- **key** (*bytes*) – key to delete
- **sync** (*bool*) – whether to use synchronous writes

write_batch(*transaction=False*, *sync=False*)
Create a new *WriteBatch* instance for this database.

See the *WriteBatch* API for more information.

Note that this method does not write a batch to the database; it only creates a new write batch instance.

Parameters

- **transaction** (*bool*) – whether to enable transaction-like behaviour when the batch is used in a `with` block
- **sync** (*bool*) – whether to use synchronous writes

Returns new *WriteBatch* instance

Return type *WriteBatch*

iterator(*reverse=False*, *start=None*, *stop=None*, *include_start=True*, *include_stop=False*, *prefix=None*, *include_key=True*, *include_value=True*, *verify_checksums=False*, *fill_cache=True*)
Create a new *Iterator* instance for this database.

All arguments are optional, and not all arguments can be used together, because some combinations make no sense. In particular:

- *start* and *stop* cannot be used if a *prefix* is specified.
- *include_start* and *include_stop* are only used if *start* and *stop* are specified.

Note: due to the whay the *prefix* support is implemented, this feature only works reliably when the default DB comparator is used.

See the *Iterator* API for more information about iterators.

Parameters

- **reverse** (*bool*) – whether the iterator should iterate in reverse order
- **start** (*bytes*) – the start key (inclusive by default) of the iterator range
- **stop** (*bytes*) – the stop key (exclusive by default) of the iterator range
- **include_start** (*bool*) – whether to include the start key in the range
- **include_stop** (*bool*) – whether to include the stop key in the range
- **prefix** (*bytes*) – prefix that all keys in the the range must have
- **include_key** (*bool*) – whether to include keys in the returned data
- **include_value** (*bool*) – whether to include values in the returned data
- **verify_checksums** (*bool*) – whether to verify checksums
- **fill_cache** (*bool*) – whether to fill the cache

Returns new *Iterator* instance

Return type *Iterator*

raw_iterator(*verify_checksums=False*, *fill_cache=True*)
Create a new *RawIterator* instance for this database.

See the *RawIterator* API for more information.

snapshot()
Create a new *Snapshot* instance for this database.

See the *Snapshot* API for more information.

get_property(*name*)
Get the specified property from LevelDB.

This returns the property value or *None* if no value is available. Example property name: `b'leveldb.stats'`.

See the description for `DB::GetProperty()` in the LevelDB C++ API for more information.

Parameters **name** (*bytes*) – name of the property

Returns property value or *None*

Return type bytes

compact_range(*start=None*, *stop=None*)
Compact underlying storage for the specified key range.

See the description for `DB::CompactRange()` in the LevelDB C++ API for more information.

Parameters

- **start** (*bytes*) – start key of range to compact (optional)
- **stop** (*bytes*) – stop key of range to compact (optional)

approximate_size(*start*, *stop*)
Return the approximate file system size for the specified range.

See the description for `DB::GetApproximateSizes()` in the LevelDB C++ API for more information.

Parameters

- **start** (*bytes*) – start key of the range

- **stop** (*bytes*) – stop key of the range

Returns approximate size

Return type int

approximate_sizes(**ranges*)
Return the approximate file system sizes for the specified ranges.

This method takes a variable number of arguments. Each argument denotes a range as a *(start, stop)* tuple, where *start* and *stop* are both byte strings. Example:

```
db.approximate_sizes(
    (b'a-key', b'other-key'),
    (b'some-other-key', b'yet-another-key'))
```

See the description for `DB::GetApproximateSizes()` in the LevelDB C++ API for more information.

Parameters **ranges** – variable number of *(start, stop)* tuples

Returns approximate sizes for the specified ranges

Return type list

prefixed_db(*prefix*)
Return a new *PrefixedDB* instance for this database.

See the *PrefixedDB* API for more information.

Parameters **prefix** (*bytes*) – prefix to use

Returns new *PrefixedDB* instance

Return type *PrefixedDB*

3.1.1 Prefixed database

class PrefixedDB
A *DB*-like object that transparently prefixes all database keys.

Do not instantiate directly; use *DB.prefixed_db()* instead.

prefix
The prefix used by this *PrefixedDB*.

db
The underlying *DB* instance.

get(...)
See *DB.get()*.

put(...)
See *DB.put()*.

delete(...)
See *DB.delete()*.

write_batch(...)
See *DB.write_batch()*.

iterator(...)
See *DB.iterator()*.

snapshot(...)
See *DB.snapshot()*.

prefixed_db(...)
Create another *PrefixedDB* instance with an additional key prefix, which will be appended to the prefix used by this *PrefixedDB* instance.

See *DB.prefixed_db()*.

3.1.2 Database maintenance

Existing databases can be repaired or destroyed using these module level functions:

repair_db(*name*, *paranoid_checks=None*, *write_buffer_size=None*, *max_open_files=None*, *lru_cache_size=None*, *block_size=None*, *block_restart_interval=None*, *max_file_size=None*, *compression='snappy'*, *bloom_filter_bits=0*, *comparator=None*, *comparator_name=None*)
Repair the specified database.

See *DB* for a description of the arguments.

See the description for `RepairDB()` in the LevelDB C++ API for more information.

destroy_db(*name*)
Destroy the specified database.

Parameters **name** (*str*) – name of the database (directory name)

See the description for `DestroyDB()` in the LevelDB C++ API for more information.

3.2 Write batch

class WriteBatch
Write batch for batch put/delete operations

Instances of this class can be used as context managers (Python's `with` block). When the `with` block terminates, the write batch will automatically write itself to the database without an explicit call to *WriteBatch.write()*:

```
with db.write_batch() as b:
    b.put(b'key', b'value')
```

The *transaction* argument to *DB.write_batch()* specifies whether the batch should be written after an exception occurred in the `with` block. By default, the batch is written (this is like a `try` statement with a `finally` clause), but if transaction mode is enabled', the batch will be discarded (this is like a `try` statement with an `else` clause).

Note: methods on a *WriteBatch* do not take a *sync* argument; this flag can be specified for the complete write batch when it is created using *DB.write_batch()*.

Do not instantiate directly; use *DB.write_batch()* instead.

See the descriptions for `WriteBatch` and `DB::Write()` in the LevelDB C++ API for more information.

put(*key*, *value*)
Set a value for the specified key.

This is like *DB.put()*, but operates on the write batch instead.

delete(*key*)
Delete the key/value pair for the specified key.

This is like *DB.delete()*, but operates on the write batch instead.

clear()
Clear the batch.

This discards all updates buffered in this write batch.

write()
Write the batch to the associated database. If you use the write batch as a context manager (in a `with` block), this method will be invoked automatically.

append(*source*)
Copy the operations in *source* (another *WriteBatch* instance) to this batch.

approximate_size()
Return the size of the database changes caused by this batch.

3.3 Snapshot

class Snapshot
Database snapshot

A snapshot provides a consistent view over keys and values. After making a snapshot, puts and deletes on the database will not be visible by the snapshot.

Do not keep unnecessary references to instances of this class around longer than needed, because LevelDB will not release the resources required for this snapshot until a snapshot is released.

Do not instantiate directly; use *DB.snapshot()* instead.

See the descriptions for `DB::GetSnapshot()` and `DB::ReleaseSnapshot()` in the LevelDB C++ API for more information.

get(...)
Get the value for the specified key, or *None* if no value was set.

Same as *DB.get()*, but operates on the snapshot instead.

iterator(...)
Create a new *Iterator* instance for this snapshot.

Same as *DB.iterator()*, but operates on the snapshot instead.

raw_iterator(...)
Create a new *RawIterator* instance for this snapshot.

Same as *DB.raw_iterator()*, but operates on the snapshot instead.

close()
Close the snapshot. Can also be accomplished using a context manager. See *Iterator.close()* for an example.

New in version 0.8.

release()
Alias for *Snapshot.close()*. *Release* is the terminology used in the LevelDB C++ API.

New in version 0.8.

3.4 Iterator

3.4.1 Regular iterators

Plyvel's *Iterator* is intended to be used like a normal Python iterator, so you can just use a standard `for` loop to iterate over it. Directly invoking methods on the *Iterator* returned by *DB.iterator()* method is only required for additional functionality.

class Iterator

Iterator to iterate over (ranges of) a database

The next item in the iterator can be obtained using the `next()` built-in or by looping over the iterator using a `for` loop.

Do not instantiate directly; use *DB.iterator()* or *Snapshot.iterator()* instead.

See the descriptions for `DB::NewIterator()` and `Iterator` in the LevelDB C++ API for more information.

prev()

Move one step back and return the previous entry.

This returns the same value as the most recent `next()` call (if any).

seek_to_start()

Move the iterator to the start key (or the begin).

This "rewinds" the iterator, so that it is in the same state as when first created. This means calling `next()` afterwards will return the first entry.

seek_to_stop()

Move the iterator to the stop key (or the end).

This "fast-forwards" the iterator past the end. After this call the iterator is exhausted, which means a call to `next()` raises StopIteration, but *prev()* will work.

seek(*target*)

Move the iterator to the specified *target*.

This moves the iterator to the the first key that sorts equal or after the specified *target* within the iterator range (*start* and *stop*).

close()

Close the iterator.

This closes the iterator and releases the associated resources. Any further operations on the closed iterator will raise `RuntimeError`.

To automatically close an iterator, a context manager can be used:

```
with db.iterator() as it:
    for k, v in it:
        pass  # do something

it.seek_to_start()  # raises RuntimeError
```

New in version 0.6.

3.4.2 Raw iterators

The raw iteration API mimics the C++ iterator interface provided by LevelDB. See the LevelDB documentation for a detailed description.

class RawIterator
Raw iterator to iterate over a database

New in version 0.7.

valid()
Check whether the iterator is currently valid.

seek_to_first()
Seek to the first key (if any).

seek_to_last()
Seek to the last key (if any).

seek(*target*)
Seek to or past the specified key (if any).

next()
Move the iterator one step forward.

May raise *IteratorInvalidError*.

prev()
Move the iterator one step backward.

May raise *IteratorInvalidError*.

key()
Return the current key.

May raise *IteratorInvalidError*.

value()
Return the current value.

May raise *IteratorInvalidError*.

item()
Return the current key and value as a tuple.

May raise *IteratorInvalidError*.

close()
Close the iterator. Can also be accomplished using a context manager. See *Iterator.close()*.

3.5 Errors

Plyvel uses standard exceptions like `TypeError`, `ValueError`, and `RuntimeError` as much as possible. For LevelDB specific errors, Plyvel may raise a few custom exceptions, which are described below.

exception Error
Generic LevelDB error

This class is also the "parent" error for other LevelDB errors (*IOError* and *CorruptionError*). Other exceptions from this module extend from this class.

exception IOError
LevelDB IO error

This class extends both the main LevelDB Error class from this module and Python's built-in IOError.

exception CorruptionError
LevelDB corruption error

exception IteratorInvalidError
Used by *RawIterator* to signal invalid iterator state.

CHAPTER 4

Version history

4.1 Plyvel 1.4.0

Release date: TBD

- The minimum LevelDB version is now 1.21
- Add support for *`WriteBatch.append()`*
- Add support for *`WriteBatch.approximate_size()`*

4.2 Plyvel 1.3.0

Release date: 2020-10-10

- Use manylinux2010 instead of manylinux1 to build wheels (pr #103)
- Add Python 3.9 support
- Drop Python 3.5 support
- Completely drop Python 2 support

4.3 Plyvel 1.2.0

Release date: 2020-01-22

- Add Python 3.8 support (pr #109)
- Drop Python 3.4 support (pr #109)
- Build Linux wheels against Snappy 1.1.8, LevelDB 1.22, and produce Python 3.8 wheels (issue #108, pr #111,)

- Improve compilation flags for Darwin (OSX) builds (pr #107)

4.4 Plyvel 1.1.0

Release date: 2019-05-02

- Expose `name` attribute to Python code (pr #90)
- Fix building sources on OSX. (issue #95, pr #97)
- Build Linux wheels against LevelDB 1.21

4.5 Plyvel 1.0.5

Release date: 2018-07-17

- Rebuild wheels: build against Snappy 1.1.7, and produce Python 3.7 wheels (issue #78, pr #79)

4.6 Plyvel 1.0.4

Release date: 2018-01-17

- Build Python wheels with Snappy compression support. (issue #68)

4.7 Plyvel 1.0.3

Release date: 2018-01-16

- Fix building sources on OSX. (issue #66, pr #67)

4.8 Plyvel 1.0.2

Release date: 2018-01-12

- Correctly build wide unicode Python 2.7 wheels (cp27-cp27mu, UCS4). (issue #65)

4.9 Plyvel 1.0.1

Release date: 2018-01-05

- Provide binary packages (manylinux1 wheels) for Linux.

  These wheel packages have the LevelDB library embedded. This should make installation on many Linux systems easier since these packages do not depend on a recent LevelDB version being installed system-wide: running `pip install` will simply download and install the extension, instead of compiling it. (pr #64, issue #62, issue #63)

4.10 Plyvel 1.0.0

Release date: 2018-01-03

- First 1.x release. This library is quite mature, so there is no reason to keep using 0.x version numbers. While at it, switch to semantic versioning.
- Drop support for older Python versions. Minimum versions are now Python 3.4+ for modern Python and Python 2.7+ for legacy Python.
- The mimimum LevelDB version is now 1.20, which added an option for the maximum file size, which is now exposed in Plyvel. (pr #61)
- The various `.put()` methods are no longer restricted to just *bytes* (*str* in Python 2), but will accept any type implementing Python's buffer protocol, such as *bytes*, *bytearray*, and *memoryview*. Note that this only applies to values; keys must still be *bytes*. (issue #52)

4.11 Plyvel 0.9

Release date: 2014-08-27

- Ensure that the Python GIL is initialized when a custom comparator is used, since the background thread LevelDB uses for compaction calls back into Python code in that case. This makes single-threaded programs using a custom comparator work as intended. (issue #35)

4.12 Plyvel 0.8

Release date: 2013-11-29

- Allow snapshots to be closed explicitly using either *`Snapshot.close()`* or a `with` block (issue #21)

4.13 Plyvel 0.7

Release date: 2013-11-15

- New raw iterator API that mimics the LevelDB C++ interface. See *`DB.raw_iterator()`* and *`RawIterator`*. (issue #17)
- Migrate to *pytest* and *tox* for testing (issue #24)
- Performance improvements in iterator and write batch construction. The internal calls within Plyvel are now a bit faster, and the *weakref* handling required for iterators is now a lot faster due to replacing `weakref.WeakValueDictionary` with manual *weakref* handling.
- The *fill_cache*, *verify_checksums*, and *sync* arguments to various methods are now correctly taken into account everywhere, and their default values are now booleans reflecting the the LevelDB defaults.

4.14 Plyvel 0.6

Release date: 2013-10-18

- Allow iterators to be closed explicitly using either *`Iterator.close()`* or a `with` block (issue #19)

- Add useful `__repr__()` for *DB* and *PrefixedDB* instances (issue #16)

4.15 Plyvel 0.5

Release date: 2013-09-17

- Fix *Iterator.seek()* for *PrefixedDB* iterators (issue #15)
- Make some argument type checking a bit stricter (mostly `None` checks)
- Support LRU caches larger than 2GB by using the right integer type for the `lru_cache_size` *DB* constructor argument.
- Documentation improvements

4.16 Plyvel 0.4

Release date: 2013-06-17

- Add optional 'default' argument for all `.get()` methods (issue #11)

4.17 Plyvel 0.3

Release date: 2013-06-03

- Fix iterator behaviour for reverse iterators using a prefix (issue #9)
- Documentation improvements

4.18 Plyvel 0.2

Release date: 2013-03-15

- Fix iterator behaviour for iterators using non-existing start or stop keys (issue #4)

4.19 Plyvel 0.1

Release date: 2012-11-26

- Initial release

CHAPTER 5

Contributing and developing

5.1 Reporting issues

Plyvel uses Github's issue tracker. See the Plyvel project page on Github.

5.2 Obtaining the source code

The Plyvel source code can be found on Github. See the Plyvel project page on Github.

5.3 Compiling from source

A simple `make` suffices to build the Plyvel extension. Note that the `setup.py` script does *not* invoke Cython, so that installations using `pip install` do not need to depend on Cython.

A few remarks about the code:

- Plyvel is mostly written in Cython. The LevelDB API is described in *leveldb.pxd*, and subsequently used from Cython.
- The custom comparator support is written in C++ since it contains a C++ class that extends a LevelDB C++ class. The Python C API is used for the callbacks into Python. This custom class is made available in Cython using *comparator.pxd*.

5.4 Running the tests

Almost all Plyvel code is covered by the unit tests. Plyvel uses *pytest* and *tox* for running those tests. Type `make test` to run the unit tests, or run `tox` to run the tests against multiple Python versions.

5.5 Producing binary packages

To build a non-portable binary package for a single platform:

```
python setup.py bdist_wheel
```

See the comments at the top of the `Dockerfile` for instructions on how to build portable `manylinux1` wheels for multiple Python versions that should work on many Linux platforms.

5.6 Generating the documentation

The documentation is written in ReStructuredText (reST) format and processed using *Sphinx*. Type `make doc` to build the HTML documentation.

CHAPTER 6

License

External links

- Online documentation (Read the docs)
- Project page with source code and issue tracker (Github)
- Python Package Index (PyPI) page with released tarballs

Index

Symbols

A

C

D

E

G

I

K

N

P

R

S

V

W