pyrocksdb Documentation

Release 0.2.1

sh

May 31, 2014

CHAPTER 1

Overview

Python bindings to the C++ interface of http://rocksdb.org/ using cython:

```
import rocksdb
db = rocksdb.DB("test.db", rocksdb.Options(create_if_missing=True))
db.put(b"a", b"b")
print db.get(b"a")
```

Tested with python2.7 and python3.3 and RocksDB version 2.8.fb

1.1 Installing

1.1.1 Building rocksdb

Briefly describes how to build rocksdb under a ordinary debian/ubuntu. For more details consider https://github.com/facebook/rocksdb/blob/master/INSTALL.md:

```
$ apt-get install build-essential
$ apt-get install libsnappy-dev zlib1g-dev libbz2-dev libgflags-dev
$ git clone https://github.com/facebook/rocksdb.git
$ git checkout 2.8.fb
$ cd rocksdb
$ make shared_lib
```

If you do not want to call `make install` export the following enviroment variables:

```
$ export CPLUS_INCLUDE_PATH=${CPLUS_INCLUDE_PATH}:`pwd`/include
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:`pwd`
$ export LIBRARY_PATH=${LIBRARY_PATH}:`pwd`
```

1.1.2 Building pyrocksdb

```
$ apt-get install python-virtualenv python-dev
$ virtualenv pyrocks_test
$ cd pyrocks_test
$ . bin/active
$ pip install "Cython>=0.20"
$ pip install git+git://github.com/stephan-hof/pyrocksdb.git@v0.2.1
```

1.2 Basic Usage of pyrocksdb

1.2.1 Open

The most basic open call is

```
import rocksdb

db = rocksdb.DB("test.db", rocksdb.Options(create_if_missing=True))
```

A more production ready open can look like this

```
import rocksdb

opts = rocksdb.Options()
opts.create_if_missing = True
opts.max_open_files = 300000
opts.write_buffer_size = 67108864
opts.max_write_buffer_number = 3
opts.target_file_size_base = 67108864
opts.filter_policy = rocksdb.BloomFilterPolicy(10)
opts.block_cache = rocksdb.LRUCache(2 * (1024 ** 3))
opts.block_cache_compressed = rocksdb.LRUCache(500 * (1024 ** 2))

db = rocksdb.DB("test.db", opts)
```

It assings a cache of 2.5G, uses a bloom filter for faster lookups and keeps more data (64 MB) in memory before writting a .sst file.

1.2.2 About Bytes And Unicode

RocksDB stores all data as uninterpreted *byte strings*. pyrocksdb behaves the same and uses nearly everywhere byte strings too. In python2 this is the `str` type. In python3 the `bytes` type. Since the default string type for string literals differs between python 2 and 3, it is strongly recommended to use an explicit `b` prefix for all byte string literals in both python2 and python3 code. For example `b'this is a byte string'`. This avoids ambiguity and ensures that your code keeps working as intended if you switch between python2 and python3.

The only place where you can pass unicode objects are filesytem paths like

- Directory name of the database itself `rocksdb.DB.__init__()`
- `rocksdb.Options.wal_dir`
- `rocksdb.Options.db_log_dir`

To encode this path name, *sys.getfilesystemencoding()* encoding is used.

1.2.3 Access

Store, Get, Delete is straight forward

```
# Store
db.put(b"key", b"value")

# Get
db.get(b"key")
```

```
# Delete
db.delete(b"key")
```

It is also possible to gather modifications and apply them in a single operation

```
batch = rocksdb.WriteBatch()
batch.put(b"key", b"v1")
batch.delete(b"key")
batch.put(b"key", b"v2")
batch.put(b"key", b"v3")

db.write(batch)
```

Fetch of multiple values at once

```
db.put(b"key1", b"v1")
db.put(b"key2", b"v2")

ret = db.multi_get([b"key1", b"key2", b"key3"])

# prints b"v1"
print ret[b"key1"]

# prints None
print ret[b"key3"]
```

1.2.4 Iteration

Iterators behave slightly different than expected. Per default they are not valid. So you have to call one of its seek methods first

```
db.put(b"key1", b"v1")
db.put(b"key2", b"v2")
db.put(b"key3", b"v3")

it = db.iterkeys()
it.seek_to_first()

# prints [b'key1', b'key2', b'key3']
print list(it)

it.seek_to_last()
# prints [b'key3']
print list(it)

it.seek(b'key2')
# prints [b'key2', b'key3']
print list(it)
```

There are also methods to iterate over values/items

```
it = db.itervalues()
it.seek_to_first()

# prints [b'v1', b'v2', b'v3']
print list(it)

it = db.iteritems()
```

```
it.seek_to_first()

# prints [(b'key1', b'v1'), (b'key2, b'v2'), (b'key3', b'v3')]
print list(it)
```

Reversed iteration

```
it = db.iteritems()
it.seek_to_last()

# prints [(b'key3', b'v3'), (b'key2', b'v2'), (b'key1', b'v1')]
print list(reversed(it))
```

1.2.5 Snapshots

Snapshots are nice to get a consistent view on the database

```
self.db.put(b"a", b"1")
self.db.put(b"b", b"2")

snapshot = self.db.snapshot()
self.db.put(b"a", b"2")
self.db.delete(b"b")

it = self.db.iteritems()
it.seek_to_first()

# prints {b'a': b'2'}
print dict(it)

it = self.db.iteritems(snapshot=snapshot)
it.seek_to_first()

# prints {b'a': b'1', b'b': b'2'}
print dict(it)
```

1.2.6 MergeOperator

Merge operators are useful for efficient read-modify-write operations. For more details see Merge Operator

A python merge operator must either implement the `rocksdb.interfaces.AssociativeMergeOperator` or `rocksdb.interfaces.MergeOperator` interface.

The following example python merge operator implements a counter

```
class AssocCounter(rocksdb.interfaces.AssociativeMergeOperator):
    def merge(self, key, existing_value, value):
        if existing_value:
            s = int(existing_value) + int(value)
            return (True, str(s).encode('ascii'))
        return (True, value)

    def name(self):
        return b'AssocCounter'
```

```
opts = rocksdb.Options()
opts.create_if_missing = True
opts.merge_operator = AssocCounter()
db = rocksdb.DB('test.db', opts)

db.merge(b"a", b"1")
db.merge(b"a", b"1")

# prints b'2'
print db.get(b"a")
```

1.2.7 PrefixExtractor

According to Prefix API a prefix_extractor can reduce IO for scans within a prefix range. A python prefix extractor must implement the `rocksdb.interfaces.SliceTransform` interface.

The following example presents a prefix extractor of a static size. So always the first 5 bytes are used as the prefix

```
class StaticPrefix(rocksdb.interfaces.SliceTransform):
    def name(self):
        return b'static'

    def transform(self, src):
        return (0, 5)

    def in_domain(self, src):
        return len(src) >= 5

    def in_range(self, dst):
        return len(dst) == 5

opts = rocksdb.Options()
opts.create_if_missing=True
opts.prefix_extractor = StaticPrefix()

db = rocksdb.DB('test.db', opts)

db.put(b'00001.x', b'x')
db.put(b'00001.y', b'y')
db.put(b'00001.z', b'z')

db.put(b'00002.x', b'x')
db.put(b'00002.y', b'y')
db.put(b'00002.z', b'z')

db.put(b'00003.x', b'x')
db.put(b'00003.y', b'y')
db.put(b'00003.z', b'z')

it = db.iteritems(prefix=b'00002')
it.seek(b'00002')

# prints {b'00002.z': b'z', b'00002.y': b'y', b'00002.x': b'x'}
print dict(it)
```

1.2.8 Backup And Restore

Backup and Restore is done with a separate `rocksdb.BackupEngine` object.

A backup can only be created on a living database object.

```
import rocksdb

db = rocksdb.DB("test.db", rocksdb.Options(create_if_missing=True))
db.put(b'a', b'v1')
db.put(b'b', b'v2')
db.put(b'c', b'v3')
```

Backup is created like this. You can choose any path for the backup destination except the db path itself. If `flush_before_backup` is `True` the current memtable is flushed to disk before backup.

```
backup = rocksdb.BackupEngine("test.db/backups")
backup.create_backup(db, flush_before_backup=True)
```

Restore is done like this. The two arguments are the db_dir and wal_dir, which are mostly the same.

```
backup = rocksdb.BackupEngine("test.db/backups")
backup.restore_latest_backup("test.db", "test.db")
```

1.2.9 Change Memtable Or SST Implementations

As noted here *MemtableFactories*, RocksDB offers different implementations for the memtable representation. Per default `rocksdb.SkipListMemtableFactory` is used, but changing it to a different one is veary easy.

Here is an example for HashSkipList-MemtableFactory. Keep in mind: To use the hashed based MemtableFactories you must set `rocksdb.Options.prefix_extractor`. In this example all keys have a static prefix of len 5.

```
class StaticPrefix(rocksdb.interfaces.SliceTransform):
    def name(self):
        return b'static'

    def transform(self, src):
        return (0, 5)

    def in_domain(self, src):
        return len(src) >= 5

    def in_range(self, dst):
        return len(dst) == 5


opts = rocksdb.Options()
opts.prefix_extractor = StaticPrefix()
opts.memtable_factory = rocksdb.HashSkipListMemtableFactory()
opts.create_if_missing = True

db = rocksdb.DB("test.db", opts)
db.put(b'00001.x', b'x')
db.put(b'00001.y', b'y')
db.put(b'00002.x', b'x')
```

For initial bulk loads the Vector-MemtableFactory makes sense.

```
opts = rocksdb.Options()
opts.memtable_factory = rocksdb.VectorMemtableFactory()
opts.create_if_missing = True

db = rocksdb.DB("test.db", opts)
```

As noted here *TableFactories*, it is also possible to change the representation of the final data files. Here is an example how to use one of the 'PlainTables'.

```
opts = rocksdb.Options()
opts.table_factory = rocksdb.TotalOrderPlainTableFactory()
opts.create_if_missing = True

db = rocksdb.DB("test.db", opts)
```

1.2.10 Change Compaction Style

RocksDB has a compaction algorithm called *universal*. This style typically results in lower write amplification but higher space amplification than Level Style Compaction. See here for more details, https://github.com/facebook/rocksdb/wiki/Rocksdb-Architecture-Guide#multi-threaded-compactions

Here is an example to switch to *universal style compaction*.

```
opts = rocksdb.Options()
opts.compaction_style = "universal"
opts.compaction_options_universal = {"min_merge_width": 3}
```

See here for more options on *universal style compaction*, `rocksdb.Options.compaction_options_universal`

1.3 Python driver for RocksDB

1.3.1 Options creation

Options object

class `rocksdb.`**`Options`**

Important: The default values mentioned here, describe the values of the C++ library only. This wrapper does not set any default value itself. So as soon as the rocksdb developers change a default value this document could be outdated. So if you really depend on a default value, double check it with the according version of the C++ library.

Most recent default values should be here

https://github.com/facebook/rocksdb/blob/master/include/rocksdb/options.h

https://github.com/facebook/rocksdb/blob/master/util/options.cc

`__init__`(***kwargs*)

All options mentioned below can also be passed as keyword-arguments in the constructor. For example:

```python
import rocksdb

opts = rocksdb.Options(create_if_missing=True)
# is the same as
opts = rocksdb.Options()
opts.create_if_missing = True
```

`create_if_missing`

If `True`, the database will be created if it is missing.

Type: `bool`

Default: `False`

`error_if_exists`

If `True`, an error is raised if the database already exists.

Type: `bool`

Default: `False`

`paranoid_checks`

If `True`, the implementation will do aggressive checking of the data it is processing and will stop early if it detects any errors. This may have unforeseen ramifications: for example, a corruption of one DB entry may cause a large number of entries to become unreadable or for the entire DB to become unopenable. If any of the writes to the database fails (Put, Delete, Merge, Write), the database will switch to read-only mode and fail all other Write operations.

Type: `bool`

Default: `True`

`write_buffer_size`

Amount of data to build up in memory (backed by an unsorted log on disk) before converting to a sorted on-disk file.

Larger values increase performance, especially during bulk loads. Up to max_write_buffer_number write buffers may be held in memory at the same time, so you may wish to adjust this parameter to control memory usage. Also, a larger write buffer will result in a longer recovery time the next time the database is opened.

Type: `int`

Default: `4194304`

`max_write_buffer_number`

The maximum number of write buffers that are built up in memory. The default is 2, so that when 1 write buffer is being flushed to storage, new writes can continue to the other write buffer.

Type: `int`

Default: `2`

`min_write_buffer_number_to_merge`

The minimum number of write buffers that will be merged together before writing to storage. If set to 1, then all write buffers are fushed to L0 as individual files and this increases read amplification because a get request has to check in all of these files. Also, an in-memory merge may result in writing lesser data to storage if there are duplicate records in each of these individual write buffers.

Type: `int`

Default: `1`

`max_open_files`

Number of open files that can be used by the DB. You may need to increase this if your database has a large working set. Value -1 means files opened are always kept open. You can estimate number of files based on target_file_size_base and target_file_size_multiplier for level-based compaction. For universal-style compaction, you can usually set it to -1.

Type: `int`

Default: `5000`

`block_cache`

Control over blocks (user data is stored in a set of blocks, and a block is the unit of reading from disk).

If not `None` use the specified cache for blocks. If `None`, rocksdb will automatically create and use an 8MB internal cache.

Type: Instace of `rocksdb.LRUCache`

Default: `None`

`block_cache_compressed`

If not `None` use the specified cache for compressed blocks. If `None`, rocksdb will not use a compressed block cache.

Type: Instace of `rocksdb.LRUCache`

Default: `None`

`block_size`

Approximate size of user data packed per block. Note that the block size specified here corresponds to uncompressed data. The actual size of the unit read from disk may be smaller if compression is enabled. This parameter can be changed dynamically.

Type: `int`

Default: `4096`

`block_restart_interval`

Number of keys between restart points for delta encoding of keys. This parameter can be changed dynamically. Most clients should leave this parameter alone.

Type: `int`

Default: `16`

`compression`

Compress blocks using the specified compression algorithm. This parameter can be changed dynamically.

Type: Member of `rocksdb.CompressionType`

Default: `rocksdb.CompressionType.snappy_compression`

`whole_key_filtering`

If `True`, place whole keys in the filter (not just prefixes). This must generally be true for gets to be efficient.

Type: `bool`

Default: `True`

`num_levels`

Number of levels for this database

Type: `int`

Default: `7`

`level0_file_num_compaction_trigger`

Number of files to trigger level-0 compaction. A value <0 means that level-0 compaction will not be triggered by number of files at all.

Type: `int`

Default: `4`

`level0_slowdown_writes_trigger`

Soft limit on number of level-0 files. We start slowing down writes at this point. A value <0 means that no writing slow down will be triggered by number of files in level-0.

Type: `int`

Default: `20`

`level0_stop_writes_trigger`

Maximum number of level-0 files. We stop writes at this point.

Type: `int`

Default: `24`

`max_mem_compaction_level`

Maximum level to which a new compacted memtable is pushed if it does not create overlap. We try to push to level 2 to avoid the relatively expensive level 0=>1 compactions and to avoid some expensive manifest file operations. We do not push all the way to the largest level since that can generate a lot of wasted disk space if the same key space is being repeatedly overwritten.

Type: `int`

Default: `2`

`target_file_size_base`

Target file size for compaction.
target_file_size_base is per-file size for level-1.
Target file size for level L can be calculated by
target_file_size_base * (target_file_size_multiplier ^ (L-1)).

For example, if target_file_size_base is 2MB and target_file_size_multiplier is 10, then each file on level-1 will be 2MB, and each file on level 2 will be 20MB, and each file on level-3 will be 200MB.

Type: `int`

Default: `2097152`

`target_file_size_multiplier`

by default target_file_size_multiplier is 1, which means
by default files in different levels will have similar size.

Type: `int`

Default: `1`

`max_bytes_for_level_base`

Control maximum total data size for a level. *max_bytes_for_level_base* is the max total for level-1. Maximum number of bytes for level L can be calculated as (*max_bytes_for_level_base*) * (*max_bytes_for_level_multiplier* ^ (L-1)) For example, if *max_bytes_for_level_base* is 20MB, and if *max_bytes_for_level_multiplier* is 10, total data size for level-1 will be 20MB, total file size for level-2 will be 200MB, and total file size for level-3 will be 2GB.

Type: `int`

Default: `10485760`

`max_bytes_for_level_multiplier`

See `max_bytes_for_level_base`

Type: `int`

Default: `10`

`max_bytes_for_level_multiplier_additional`

Different max-size multipliers for different levels. These are multiplied by max_bytes_for_level_multiplier to arrive at the max-size of each level.

Type: `[int]`

Default: `[1, 1, 1, 1, 1, 1, 1]`

`expanded_compaction_factor`

Maximum number of bytes in all compacted files. We avoid expanding the lower level file set of a compaction if it would make the total compaction cover more than (expanded_compaction_factor * targetFileSizeLevel()) many bytes.

Type: `int`

Default: `25`

`source_compaction_factor`

Maximum number of bytes in all source files to be compacted in a single compaction run. We avoid picking too many files in the source level so that we do not exceed the total source bytes for compaction to exceed (source_compaction_factor * targetFileSizeLevel()) many bytes. If 1 pick maxfilesize amount of data as the source of a compaction.

Type: `int`

Default: `1`

`max_grandparent_overlap_factor`

Control maximum bytes of overlaps in grandparent (i.e., level+2) before we stop building a single file in a level->level+1 compaction.

Type: `int`

Default: `10`

`disable_data_sync`

If true, then the contents of data files are not synced to stable storage. Their contents remain in the OS buffers till the OS decides to flush them. This option is good for bulk-loading of data. Once the bulk-loading is complete, please issue a sync to the OS to flush all dirty buffesrs to stable storage.

Type: `bool`

Default: `False`

`use_fsync`

If true, then every store to stable storage will issue a fsync. If false, then every store to stable storage will issue a fdatasync. This parameter should be set to true while storing data to filesystem like ext3 that can lose files after a reboot.

Type: `bool`

Default: `False`

`db_stats_log_interval`

This number controls how often a new scribe log about db deploy stats is written out. -1 indicates no logging at all.

Type: `int`

Default: `1800`

`db_log_dir`

This specifies the info LOG dir. If it is empty, the log files will be in the same dir as data. If it is non empty, the log files will be in the specified dir, and the db data dir's absolute path will be used as the log file name's prefix.

Type: `unicode`

Default: `""`

`wal_dir`

This specifies the absolute dir path for write-ahead logs (WAL). If it is empty, the log files will be in the same dir as data, dbname is used as the data dir by default. If it is non empty, the log files will be in kept the specified dir. When destroying the db, all log files in wal_dir and the dir itself is deleted

Type: `unicode`

Default: `""`

`disable_seek_compaction`

Disable compaction triggered by seek. With bloomfilter and fast storage, a miss on one level is very cheap if the file handle is cached in table cache (which is true if max_open_files is large).

Type: `bool`

Default: `True`

`delete_obsolete_files_period_micros`

The periodicity when obsolete files get deleted. The default value is 6 hours. The files that get out of scope by compaction process will still get automatically delete on every compaction, regardless of this setting

Type: `int`

Default: `21600000000`

`max_background_compactions`

Maximum number of concurrent background jobs, submitted to the default LOW priority thread pool

Type: `int`

Default: `1`

`max_background_flushes`

Maximum number of concurrent background memtable flush jobs, submitted to the HIGH priority thread pool. By default, all background jobs (major compaction and memtable flush) go to the LOW priority pool. If this option is set to a positive number, memtable flush jobs will be submitted to the HIGH priority pool. It is important when the same Env is shared by multiple db instances. Without a separate pool, long running major compaction jobs could potentially block memtable flush jobs of other db instances, leading to unnecessary Put stalls.

Type: `int`

Default: `1`

`max_log_file_size`

Specify the maximal size of the info log file. If the log file is larger than *max_log_file_size*, a new info log file will be created. If max_log_file_size == 0, all logs will be written to one log file.

Type: `int`

Default: `0`

`log_file_time_to_roll`

Time for the info log file to roll (in seconds). If specified with non-zero value, log file will be rolled if it has been active longer than *log_file_time_to_roll*. A value of `0` means disabled.

Type: `int`

Default: `0`

`keep_log_file_num`

Maximal info log files to be kept.

Type: `int`

Default: `1000`

`soft_rate_limit`

Puts are delayed 0-1 ms when any level has a compaction score that exceeds soft_rate_limit. This is ignored when == 0.0. CONSTRAINT: soft_rate_limit <= hard_rate_limit. If this constraint does not hold, RocksDB will set soft_rate_limit = hard_rate_limit. A value of `0` means disabled.

Type: `float`

Default: `0`

`hard_rate_limit`

Puts are delayed 1ms at a time when any level has a compaction score that exceeds hard_rate_limit. This is ignored when <= 1.0. A value fo `0` means disabled.

Type: `float`

Default: `0`

`rate_limit_delay_max_milliseconds`

Max time a put will be stalled when hard_rate_limit is enforced. If 0, then there is no limit.

Type: `int`

Default: `1000`

`max_manifest_file_size`

manifest file is rolled over on reaching this limit. The older manifest file be deleted. The default value is MAX_INT so that roll-over does not take place.

Type: `int`

Default: `(2**64) - 1`

`no_block_cache`

Disable block cache. If this is set to true, then no block cache should be used, and the block_cache should point to `None`

Type: `bool`

Default: `False`

`table_cache_numshardbits`

Number of shards used for table cache.

Type: `int`

Default: `4`

`table_cache_remove_scan_count_limit`

During data eviction of table's LRU cache, it would be inefficient to strictly follow LRU because this piece of memory will not really be released unless its refcount falls to zero. Instead, make two passes: the first pass will release items with refcount = 1, and if not enough space releases after scanning the number of elements specified by this parameter, we will remove items in LRU order.

Type: `int`

Default: `16`

`arena_block_size`

size of one block in arena memory allocation. If <= 0, a proper value is automatically calculated (usually 1/10 of writer_buffer_size).

Type: `int`

Default: `0`

`disable_auto_compactions`

Disable automatic compactions. Manual compactions can still be issued on this database.

Type: `bool`

Default: `False`

`wal_ttl_seconds, wal_size_limit_mb`

The following two fields affect how archived logs will be deleted.

1. If both set to 0, logs will be deleted asap and will not get into the archive.
2. If wal_ttl_seconds is 0 and wal_size_limit_mb is not 0, WAL files will be checked every 10 min and if total size is greater then wal_size_limit_mb, they will be deleted starting with the earliest until size_limit is met. All empty files will be deleted.
3. If wal_ttl_seconds is not 0 and wal_size_limit_mb is 0, then WAL files will be checked every wal_ttl_secondsi / 2 and those that are older than wal_ttl_seconds will be deleted.
4. If both are not 0, WAL files will be checked every 10 min and both checks will be performed with ttl being first.

Type: `int`

Default: `0`

`manifest_preallocation_size`

Number of bytes to preallocate (via fallocate) the manifest files. Default is 4mb, which is reasonable to reduce random IO as well as prevent overallocation for mounts that preallocate large amounts of data (such as xfs's allocsize option).

Type: `int`

Default: `4194304`

`purge_redundant_kvs_while_flush`

Purge duplicate/deleted keys when a memtable is flushed to storage.

Type: `bool`

Default: `True`

`allow_os_buffer`

Data being read from file storage may be buffered in the OS

Type: `bool`

Default: `True`

`allow_mmap_reads`

Allow the OS to mmap file for reading sst tables

Type: `bool`

Default: `True`

`allow_mmap_writes`

Allow the OS to mmap file for writing

Type: `bool`

Default: `False`

`is_fd_close_on_exec`

Disable child process inherit open files

Type: `bool`

Default: `True`

`skip_log_error_on_recovery`

Skip log corruption error on recovery (If client is ok with losing most recent changes)

Type: `bool`

Default: `False`

`stats_dump_period_sec`
If not zero, dump rocksdb.stats to LOG every stats_dump_period_sec

Type: `int`
Default: `3600`

`block_size_deviation`
This is used to close a block before it reaches the configured 'block_size'. If the percentage of free space in the current block is less than this specified number and adding a new record to the block will exceed the configured block size, then this block will be closed and the new record will be written to the next block.

Type: `int`
Default: `10`

`advise_random_on_open`
If set true, will hint the underlying file system that the file access pattern is random, when a sst file is opened.

Type: `bool`
Default: `True`

`use_adaptive_mutex`
Use adaptive mutex, which spins in the user space before resorting to kernel. This could reduce context switch when the mutex is not heavily contended. However, if the mutex is hot, we could end up wasting spin time.

Type: `bool`
Default: `False`

`bytes_per_sync`
Allows OS to incrementally sync files to disk while they are being written, asynchronously, in the background. Issue one request for every bytes_per_sync written. 0 turns it off.

Type: `int`
Default: `0`

`verify_checksums_in_compaction`
If `True`, compaction will verify checksum on every read that happens as part of compaction.

Type: `bool`
Default: `True`

compaction_style

The compaction style. Could be set to "level" to use level-style compaction. For universal-style compaction use "universal".

Type: string
Default: level

compaction_options_universal

Options to use for universal-style compaction. They make only sense if rocksdb.Options.compaction_style is set to "universal".

It is a dict with the following keys.

- **size_ratio:** Percentage flexibilty while comparing file size. If the candidate file(s) size is 1% smaller than the next file's size, then include next file into this candidate set. Default: 1
- **min_merge_width:** The minimum number of files in a single compaction run. Default: 2
- **max_merge_width:** The maximum number of files in a single compaction run. Default: UINT_MAX
- **max_size_amplification_percent:** The size amplification is defined as the amount (in percentage) of additional storage needed to store a single byte of data in the database. For example, a size amplification of 2% means that a database that contains 100 bytes of user-data may occupy upto 102 bytes of physical storage. By this definition, a fully compacted database has a size amplification of 0%. Rocksdb uses the following heuristic to calculate size amplification: it assumes that all files excluding the earliest file contribute to the size amplification. Default: 200, which means that a 100 byte database could require upto 300 bytes of storage.
- **compression_size_percent:** If this option is set to be -1 (the default value), all the output files will follow compression type specified.

  If this option is not negative, we will try to make sure compressed size is just above this value. In normal cases, at least this percentage of data will be compressed.

  When we are compacting to a new file, here is the criteria whether it needs to be compressed: assuming here are the list of files sorted by generation time: A1...An B1...Bm C1...Ct where A1 is the newest and Ct is the oldest, and we are going to compact B1...Bm, we calculate the total size of all the files as total_size, as well as the total size of C1...Ct as total_C, the compaction output file will be compressed if total_C / total_size < this percentage. Default: -1
- **stop_style:** The algorithm used to stop picking files into a single compaction. Can be either "similar_size" or "total_size".
  - similar_size: Pick files of similar size.
  - total_size: Total size of picked files is greater than next file.

  Default: "total_size"

For setting options, just assign a dict with the fields to set. It is allowed to omit keys in this dict. Missing keys are just not set to the underlying options object.

This example just changes the stop_style and leaves the other options untouched.

```
opts = rocksdb.Options()
opts.compaction_options_universal = {'stop_style': 'similar_size'}
```

filter_deletes

Use KeyMayExist API to filter deletes when this is true. If KeyMayExist returns false, i.e. the key definitely does not exist, then the delete is a noop. KeyMayExist only incurs in-memory look up. This optimization avoids writing the delete to storage when appropriate.

Type: `bool`

Default: `False`

max_sequential_skip_in_iterations

An iteration->Next() sequentially skips over keys with the same user-key unless this option is set. This number specifies the number of keys (with the same userkey) that will be sequentially skipped before a reseek is issued.

Type: `int`

Default: `8`

memtable_factory

This is a factory that provides MemTableRep objects. Right now you can assing instances of the following classes.

- `rocksdb.VectorMemtableFactory`
- `rocksdb.SkipListMemtableFactory`
- `rocksdb.HashSkipListMemtableFactory`
- `rocksdb.HashLinkListMemtableFactory`

Default: `rocksdb.SkipListMemtableFactory`

table_factory

Factory for the files forming the persisten data storage. Sometimes they are also named SST-Files. Right now you can assign instances of the following classes.

- `rocksdb.BlockBasedTableFactory`
- `rocksdb.PlainTableFactory`
- `rocksdb.TotalOrderPlainTableFactory`

Default: `rocksdb.BlockBasedTableFactory`

inplace_update_support

Allows thread-safe inplace updates. Requires Updates if

- key exists in current memtable
- new sizeof(new_value) <= sizeof(old_value)
- old_value for that key is a put i.e. kTypeValue

Type: `bool`

Default: `False`

`inplace_update_num_locks`

Number of locks used for inplace update.
Default: 10000, if inplace_update_support = true, else 0.

Type: `int`
Default: `10000`

`comparator`

Comparator used to define the order of keys in the table. A python comparator must implement the `rocksdb.interfaces.Comparator` interface.

Requires: The client must ensure that the comparator supplied here has the same name and orders keys *exactly* the same as the comparator provided to previous open calls on the same DB.

Default: `rocksdb.BytewiseComparator`

`merge_operator`

The client must provide a merge operator if Merge operation needs to be accessed. Calling Merge on a DB without a merge operator would result in `rocksdb.errors.NotSupported`. The client must ensure that the merge operator supplied here has the same name and *exactly* the same semantics as the merge operator provided to previous open calls on the same DB. The only exception is reserved for upgrade, where a DB previously without a merge operator is introduced to Merge operation for the first time. It's necessary to specify a merge operator when openning the DB in this case.

A python merge operator must implement the `rocksdb.interfaces.MergeOperator` or `rocksdb.interfaces.AssociativeMergeOperator` interface.

Default: `None`

`filter_policy`

If not `None` use the specified filter policy to reduce disk reads. A python filter policy must implement the `rocksdb.interfaces.FilterPolicy` interface. Recommendes is a instance of `rocksdb.BloomFilterPolicy`

Default: `None`

`prefix_extractor`

If not `None`, use the specified function to determine the prefixes for keys. These prefixes will be placed in the filter. Depending on the workload, this can reduce the number of read-IOP cost for scans when a prefix is passed to the calls generating an iterator (`rocksdb.DB.iterkeys()` ...).

A python prefix_extractor must implement the `rocksdb.interfaces.SliceTransform` interface

For prefix filtering to work properly, "prefix_extractor" and "comparator" must be such that the following properties hold:

1. `key.starts_with(prefix(key))`
2. `compare(prefix(key), key) <= 0`
3. `If compare(k1, k2) <= 0, then compare(prefix(k1), prefix(k2)) <= 0`
4. `prefix(prefix(key)) == prefix(key)`

Default: `None`

CompressionTypes

class `rocksdb.`**`CompressionType`**

Defines the support compression types

`no_compression`

`snappy_compression`

`zlib_compression`

`bzip2_compression`

BytewiseComparator

class `rocksdb.`**`BytewiseComparator`**

Wraps the rocksdb Bytewise Comparator, it uses lexicographic byte-wise ordering

BloomFilterPolicy

class `rocksdb.`**`BloomFilterPolicy`**

Wraps the rocksdb BloomFilter Policy

`__init__`(*bits_per_key*)

Parameters **bits_per_key** (*int*) – Specifies the approximately number of bits per key. A good value for bits_per_key is 10, which yields a filter with ~ 1% false positive rate.

LRUCache

class `rocksdb.`**`LRUCache`**

Wraps the rocksdb LRUCache

`__init__`(*capacity*, *shard_bits=None*, *rm_scan_count_limit=None*)

Create a new cache with a fixed size capacity. The cache is sharded to 2^numShardBits shards, by hash of the key. The total capacity is divided and evenly assigned to each shard. Inside each shard, the eviction is done in two passes: first try to free spaces by evicting entries that are among the most least used removeScanCountLimit entries and do not have reference other than by the cache itself, in the least-used order. If not enough space is freed, further free the entries in least used order.

TableFactories

Currently RocksDB supports two types of tables: plain table and block-based table. Instances of this classes can assigned to `rocksdb.Options.table_factory`

- *Block-based table:* This is the default table type that RocksDB inherited from LevelDB. It was designed for storing data in hard disk or flash device.
- *Plain table:* It is one of RocksDB's SST file format optimized for low query latency on pure-memory or really low-latency media.

Tutorial of rocksdb table formats is available here: https://github.com/facebook/rocksdb/wiki/A-Tutorial-of-RocksDB-SST-formats

class `rocksdb.`**`BlockBasedTableFactory`**

Wraps BlockBasedTableFactory of RocksDB.

class `rocksdb.`**`PlainTableFactory`**

Plain Table with prefix-only seek. It wraps rocksdb PlainTableFactory.

For this factory, you need to set `rocksdb.Options.prefix_extractor` properly to make it work. Look-up will start with prefix hash lookup for key prefix. Inside the hash bucket found, a binary search is executed for hash conflicts. Finally, a linear search is used.

`__init__`(*user_key_len=0*, *bloom_bits_per_prefix=10*, *hash_table_ratio=0.75*, *index_sparseness=10*)

Parameters

- **user_key_len** (*int*) – Plain table has optimization for fix-sized keys, which can be specified via user_key_len. Alternatively, you can pass *0* if your keys have variable lengths.
- **bloom_bits_per_key** (*int*) – The number of bits used for bloom filer per prefix. You may disable it by passing *0*.
- **hash_table_ratio** (*float*) – The desired utilization of the hash table used for prefix hashing. hash_table_ratio = number of prefixes / #buckets in the hash table.
- **index_sparseness** (*int*) – Inside each prefix, need to build one index record for how many keys for binary search inside each hash bucket.

class `rocksdb.`**`TotalOrderPlainTableFactory`**

This factory of plain table ignores Options.prefix_extractor and assumes no hashable prefix available to the key structure. Lookup will be based on binary search index only. Total order seek() can be issued.

`__init__`(*user_key_len=0*, *bloom_bits_per_key=0*, *index_sparseness=16*)

Parameters

- **user_key_len** (*int*) – Plain table has optimization for fix-sized keys, which can be specified via user_key_len. Alternatively, you can pass *0* if your keys have variable lengths.
- **bloom_bits_per_key** (*int*) – The number of bits used for bloom filer per key. You may disable it by passing a zero.
- **index_sparseness** (*int*) – Need to build one index record for how many keys for binary search.

MemtableFactories

RocksDB has different classes to represent the in-memory buffer for the current operations. You have to assing instances of the following classes to `rocksdb.Options.memtable_factory`. This page has a comparison the most popular ones. https://github.com/facebook/rocksdb/wiki/Hash-based-memtable-implementations

class `rocksdb.`**`VectorMemtableFactory`**

This creates MemTableReps that are backed by an std::vector. On iteration, the vector is sorted. This is useful for workloads where iteration is very rare and writes are generally not issued after reads begin.

`__init__`(*count=0*)

Parameters **count** (*int*) – Passed to the constructor of the underlying std::vector of each VectorRep. On initialization, the underlying array will be at least count bytes reserved for usage.

class `rocksdb.`**`SkipListMemtableFactory`**

This uses a skip list to store keys.

`__init__`()

class `rocksdb.`**`HashSkipListMemtableFactory`**

This class contains a fixed array of buckets, each pointing to a skiplist (null if the bucket is empty).

Note: `rocksdb.Options.prefix_extractor` must be set, otherwise rocksdb fails back to skip-list.

`__init__`(*bucket_count = 1000000*, *skiplist_height = 4*, *skiplist_branching_factor = 4*)

Parameters

- **bucket_count** (*int*) – number of fixed array buckets
- **skiplist_height** (*int*) – the max height of the skiplist
- **skiplist_branching_factor** (*int*) – probabilistic size ratio between adjacent link lists in the skiplist

class `rocksdb.`**`HashLinkListMemtableFactory`**

The factory is to create memtables with a hashed linked list. It contains a fixed array of buckets, each pointing to a sorted single linked list (null if the bucket is empty).

Note: `rocksdb.Options.prefix_extractor` must be set, otherwise rocksdb fails back to skip-list.

`__init__`(*bucket_count=50000*)

Parameters **bucket** (*int*) – number of fixed array buckets

1.3.2 Database interactions

Database object

class `rocksdb.`**`DB`**

`__init__`(*db_name*, *Options opts*, *read_only=False*)

Parameters

- **db_name** (*unicode*) – Name of the database to open
- **opts** (`rocksdb.Options`) – Options for this specific database
- **read_only** (*bool*) – If `True` the database is opened read-only. All DB calls which modify data will raise an Exception.

`put`(*key*, *value*, *sync=False*, *disable_wal=False*)

Set the database entry for "key" to "value".

Parameters

- **key** (*bytes*) – Name for this entry
- **value** (*bytes*) – Data for this entry
- **sync** (*bool*) – If `True`, the write will be flushed from the operating system buffer cache (by calling WritableFile::Sync()) before the write is considered complete. If this flag is true, writes will be slower.

  If this flag is `False`, and the machine crashes, some recent writes may be lost. Note that if it is just the process that crashes (i.e., the machine does not reboot), no writes will be lost even if `sync == False`.

In other words, a DB write with `sync == False` has similar crash semantics as the "write()" system call. A DB write with `sync == True` has similar crash semantics to a "write()" system call followed by "fdatasync()".

- **disable_wal** (*bool*) – If `True`, writes will not first go to the write ahead log, and the write may got lost after a crash.

`delete`(*key*, *sync=False*, *disable_wal=False*)
Remove the database entry for "key".

Parameters

- **key** (*bytes*) – Name to delete
- **sync** – See `rocksdb.DB.put()`
- **disable_wal** – See `rocksdb.DB.put()`

Raises **rocksdb.errors.NotFound** If the key did not exists

`merge`(*key*, *value*, *sync=False*, *disable_wal=False*)
Merge the database entry for "key" with "value". The semantics of this operation is determined by the user provided merge_operator when opening DB.

See `rocksdb.DB.put()` for the parameters

Raises `rocksdb.errors.NotSupported` if this is called and no `rocksdb.Options.merge_operator` was set at creation

`write`(*batch*, *sync=False*, *disable_wal=False*)
Apply the specified updates to the database.

Parameters

- **batch** (*rocksdb.WriteBatch*) – Batch to apply
- **sync** – See `rocksdb.DB.put()`
- **disable_wal** – See `rocksdb.DB.put()`

`get`(*key*, *verify_checksums=False*, *fill_cache=True*, *prefix_seek=False*, *snapshot=None*, *read_tier="all"*)

Parameters

- **key** (*bytes*) – Name to get
- **verify_checksums** (*bool*) – If `True`, all data read from underlying storage will be verified against corresponding checksums.
- **fill_cache** (*bool*) – Should the "data block", "index block" or "filter block" read for this iteration be cached in memory? Callers may wish to set this field to `False` for bulk scans.
- **prefix_seek** (*bool*) – If this option is set and memtable implementation allows. Seek might only return keys with the same prefix as the seek-key
- **snapshot** (`rocksdb.Snapshot`) – If not `None`, read as of the supplied snapshot (which must belong to the DB that is being read and which must not have been released). Is it `None` a implicit snapshot of the state at the beginning of this read operation is used
- **read_tier** (*string*) – Specify if this read request should process data that ALREADY resides on a particular cache. If the required data is not found at the specified cache, then `rocksdb.errors.Incomplete` is raised.

  Use `all` if a fetch from disk is allowed.

Use `cache` if only data from cache is allowed.

Returns `None` if not found, else the value for this key

multi_get(*keys*, *verify_checksums=False*, *fill_cache=True*, *prefix_seek=False*, *snapshot=None*, *read_tier="all"*)

Parameters **keys** (*list of bytes*) – Keys to fetch

For the other params see `rocksdb.DB.get()`

Returns A `dict` where the value is either `bytes` or `None` if not found

Raises If the fetch for a single key fails

Note: keys will not be "de-duplicated". Duplicate keys will return duplicate values in order.

key_may_exist(*key*, *fetch=False*, *verify_checksums=False*, *fill_cache=True*, *prefix_seek=False*, *snapshot=None*, *read_tier="all"*)

If the key definitely does not exist in the database, then this method returns `False`, else `True`. If the caller wants to obtain value when the key is found in memory, fetch should be set to `True`. This check is potentially lighter-weight than invoking DB::get(). One way to make this lighter weight is to avoid doing any IOs.

Parameters

- **key** (*bytes*) – Key to check
- **fetch** (*bool*) – Obtain also the value if found

For the other params see `rocksdb.DB.get()`

Returns

- `(True, None)` if key is found but value not in memory
- `(True, None)` if key is found and `fetch=False`
- `(True, <data>)` if key is found and value in memory and `fetch=True`
- `(False, None)` if key is not found

iterkeys(*prefix=None*, *fetch=False*, *verify_checksums=False*, *fill_cache=True*, *prefix_seek=False*, *snapshot=None*, *read_tier="all"*)

Iterate over the keys

Parameters **prefix** (*bytes*) – Not implemented yet

For other params see `rocksdb.DB.get()`

Returns A iterator object which is not valid yet. Call first one of the seek methods of the iterator to position it

Return type `rocksdb.BaseIterator`

itervalues(*prefix=None*, *fetch=False*, *verify_checksums=False*, *fill_cache=True*, *prefix_seek=False*, *snapshot=None*, *read_tier="all"*)

Iterate over the values

Parameters **prefix** (*bytes*) – Not implemented yet

For other params see `rocksdb.DB.get()`

Returns A iterator object which is not valid yet. Call first one of the seek methods of the iterator to position it

Return type `rocksdb.BaseIterator`

`iteritems`(*prefix=None*, *fetch=False*, *verify_checksums=False*, *fill_cache=True*, *prefix_seek=False*, *snapshot=None*, *read_tier="all"*)

Iterate over the items

Parameters **prefix** (*bytes*) – Not implemented yet

For other params see `rocksdb.DB.get()`

Returns A iterator object which is not valid yet. Call first one of the seek methods of the iterator to position it

Return type `rocksdb.BaseIterator`

`snapshot`()

Return a handle to the current DB state. Iterators created with this handle will all observe a stable snapshot of the current DB state.

Return type `rocksdb.Snapshot`

`get_property`(*prop*)

DB implementations can export properties about their state via this method. If "property" is a valid property understood by this DB implementation, a byte string with its value is returned. Otherwise `None`

Valid property names include:

- **`b"rocksdb.num-files-at-level<N>"`: return the number of files at level <N>,** where <N> is an ASCII representation of a level number (e.g. "0").
- **`b"rocksdb.stats"`: returns a multi-line byte string that describes statistics** about the internal operation of the DB.
- **`b"rocksdb.sstables"`: returns a multi-line byte string that describes all** of the sstables that make up the db contents.
- `b"rocksdb.num-immutable-mem-table"`: Number of immutable mem tables.
- `b"rocksdb.mem-table-flush-pending"`: Returns `1` if mem table flush is pending, otherwise `0`.
- `b"rocksdb.compaction-pending"`: Returns `1` if a compaction is pending, otherweise `0`.
- `b"rocksdb.background-errors"`: Returns accumulated background errors encountered.
- `b"rocksdb.cur-size-active-mem-table"`: Returns current size of the active memtable.

`get_live_files_metadata`()

Returns a list of all table files.

It returns a list of dict's were each dict has the following keys.

`name` Name of the file

`level` Level at which this file resides

`size` File size in bytes

`smallestkey` Smallest user defined key in the file

`largestkey` Largest user defined key in the file

`smallest_seqno` smallest seqno in file

`largest_seqno` largest seqno in file

compact_range(*begin=None*, *end=None*, *reduce_level=False*, *target_level=-1*)

Compact the underlying storage for the key range [begin,end]. The actual compaction interval might be superset of [begin, end]. In particular, deleted and overwritten versions are discarded, and the data is rearranged to reduce the cost of operations needed to access the data.

This operation should typically only be invoked by users who understand the underlying implementation.

`begin == None` is treated as a key before all keys in the database. `end == None` is treated as a key after all keys in the database. Therefore the following call will compact the entire database: `db.compact_range()`.

Note that after the entire database is compacted, all data are pushed down to the last level containing any data. If the total data size after compaction is reduced, that level might not be appropriate for hosting all the files. In this case, client could set reduce_level to `True`, to move the files back to the minimum level capable of holding the data set or a given level (specified by non-negative target_level).

Parameters

- **begin** (*bytes*) – Key where to start compaction. If `None` start at the beginning of the database.
- **end** (*bytes*) – Key where to end compaction. If `None` end at the last key of the database.
- **reduce_level** (*bool*) – If `True` allow rocksdb to move the data to another level, if the current is not big enouth. If `False` you may end with a bigger level than configured.
- **target_level** (*int*) – Level where to push the the range to compact.

options

Returns the associated `rocksdb.Options` instance.

Note: Changes to this object have no effect anymore. Consider this as read-only

Iterator

class rocksdb.**BaseIterator**

Base class for all iterators in this module. After creation a iterator is invalid. Call one of the seek methods first before starting iteration

seek_to_first()

Position at the first key in the source

seek_to_last()

Position at the last key in the source

seek(*key*)

Parameters **key** (*bytes*) – Position at the first key in the source that at or past

Methods to support the python iterator protocol

__iter__()

__next__()

__reversed__()

Snapshot

class `rocksdb.`**`Snapshot`**

Opaque handler for a single Snapshot. Snapshot is released if nobody holds a reference on it. Retrieved via `rocksdb.DB.snapshot()`

WriteBatch

class `rocksdb.`**`WriteBatch`**

WriteBatch holds a collection of updates to apply atomically to a DB.

The updates are applied in the order in which they are added to the WriteBatch. For example, the value of "key" will be "v3" after the following batch is written:

```
batch = rocksdb.WriteBatch()
batch.put(b"key", b"v1")
batch.delete(b"key")
batch.put(b"key", b"v2")
batch.put(b"key", b"v3")
```

`__init__`(*data=None*)

Creates a WriteBatch.

Parameters **data** (*bytes*) – A serialized version of a previous WriteBatch. As retrieved from a previous .data() call. If `None` a empty WriteBatch is generated

`put`(*key*, *value*)

Store the mapping "key->value" in the database.

Parameters

- **key** (*bytes*) – Name of the entry to store
- **value** (*bytes*) – Data of this entry

`merge`(*key*, *value*)

Merge "value" with the existing value of "key" in the database.

Parameters

- **key** (*bytes*) – Name of the entry to merge
- **value** (*bytes*) – Data to merge

`delete`(*key*)

If the database contains a mapping for "key", erase it. Else do nothing.

Parameters **key** (*bytes*) – Key to erase

`clear`()

Clear all updates buffered in this batch.

`data`()

Retrieve the serialized version of this batch.

Return type `bytes`

`count`()

Returns the number of updates in the batch

Return type int

Errors

exception `rocksdb.errors.`**`NotFound`**

exception `rocksdb.errors.`**`Corruption`**

exception `rocksdb.errors.`**`NotSupported`**

exception `rocksdb.errors.`**`InvalidArgument`**

exception `rocksdb.errors.`**`RocksIOError`**

exception `rocksdb.errors.`**`MergeInProgress`**

exception `rocksdb.errors.`**`Incomplete`**

1.3.3 Interfaces

Comparator

class `rocksdb.interfaces.`**`Comparator`**

A Comparator object provides a total order across slices that are used as keys in an sstable or a database. A Comparator implementation must be thread-safe since rocksdb may invoke its methods concurrently from multiple threads.

`compare`(*a*, *b*)

Three-way comparison.

Parameters

- **a** (*bytes*) – First field to compare
- **b** (*bytes*) – Second field to compare

Returns

- -1 if a < b
- 0 if a == b
- 1 if a > b

Return type `int`

`name`()

The name of the comparator. Used to check for comparator mismatches (i.e., a DB created with one comparator is accessed using a different comparator).

The client of this package should switch to a new name whenever the comparator implementation changes in a way that will cause the relative ordering of any two keys to change.

Names starting with "rocksdb." are reserved and should not be used by any clients of this package.

Return type `bytes`

Merge Operator

Essentially, a MergeOperator specifies the SEMANTICS of a merge, which only client knows. It could be numeric addition, list append, string concatenation, edit data structure, whatever. The library, on the other hand, is concerned with the exercise of this interface, at the right time (during get, iteration, compaction...)

To use merge, the client needs to provide an object implementing one of the following interfaces:

- AssociativeMergeOperator - for most simple semantics (always take two values, and merge them into one value, which is then put back into rocksdb). numeric addition and string concatenation are examples.
- MergeOperator - the generic class for all the more complex operations. One method (FullMerge) to merge a Put/Delete value with a merge operand. Another method (PartialMerge) that merges two operands together. This is especially useful if your key values have a complex structure but you would still like to support client-specific incremental updates.

AssociativeMergeOperator is simpler to implement. MergeOperator is simply more powerful.

See this page for more details https://github.com/facebook/rocksdb/wiki/Merge-Operator

AssociativeMergeOperator

class `rocksdb.interfaces.`**`AssociativeMergeOperator`**

`merge`(*key*, *existing_value*, *value*)
Gives the client a way to express the read -> modify -> write semantics

Parameters

- **key** (*bytes*) – The key that's associated with this merge operation
- **existing_value** (*bytes*) – The current value in the db. `None` indicates the key does not exist before this op
- **value** (*bytes*) – The value to update/merge the existing_value with

Returns `True` and the new value on success. All values passed in will be client-specific values. So if this method returns false, it is because client specified bad data or there was internal corruption. The client should assume that this will be treated as an error by the library.

Return type `(bool, bytes)`

`name`()
The name of the MergeOperator. Used to check for MergeOperator mismatches. For example a DB created with one MergeOperator is accessed using a different MergeOperator.

Return type `bytes`

MergeOperator

class `rocksdb.interfaces.`**`MergeOperator`**

`full_merge`(*key*, *existing_value*, *operand_list*)
Gives the client a way to express the read -> modify -> write semantics

Parameters

- **key** (*bytes*) – The key that's associated with this merge operation. Client could multiplex the merge operator based on it if the key space is partitioned and different subspaces refer to different types of data which have different merge operation semantics
- **existing_value** (*bytes*) – The current value in the db. `None` indicates the key does not exist before this op
- **operand_list** (*list of bytes*) – The sequence of merge operations to apply.

Returns `True` and the new value on success. All values passed in will be client-specific values. So if this method returns false, it is because client specified bad data or there was internal corruption. The client should assume that this will be treated as an error by the library.

Return type `(bool, bytes)`

partial_merge(*key*, *left_operand*, *right_operand*)

This function performs merge(left_op, right_op) when both the operands are themselves merge operation types that you would have passed to a DB::Merge() call in the same order. For example DB::Merge(key,left_op), followed by DB::Merge(key,right_op)).

PartialMerge should combine them into a single merge operation that is returned together with `True` This new value should be constructed such that a call to DB::Merge(key, new_value) would yield the same result as a call to DB::Merge(key, left_op) followed by DB::Merge(key, right_op).

If it is impossible or infeasible to combine the two operations, return `(False, None)` The library will internally keep track of the operations, and apply them in the correct order once a base-value (a Put/Delete/End-of-Database) is seen.

Parameters

- **key** (*bytes*) – the key that is associated with this merge operation.
- **left_operand** (*bytes*) – First operand to merge
- **right_operand** (*bytes*) – Second operand to merge

Return type `(bool, bytes)`

Note: Presently there is no way to differentiate between error/corruption and simply "return false". For now, the client should simply return false in any case it cannot perform partial-merge, regardless of reason. If there is corruption in the data, handle it in the FullMerge() function, and return false there.

name()

The name of the MergeOperator. Used to check for MergeOperator mismatches. For example a DB created with one MergeOperator is accessed using a different MergeOperator.

Return type `bytes`

FilterPolicy

class `rocksdb.interfaces.`**FilterPolicy**

create_filter(*keys*)

Create a bytestring which can act as a filter for keys.

Parameters **keys** (*list of bytes*) – list of keys (potentially with duplicates) that are ordered according to the user supplied comparator.

Returns A filter that summarizes keys

Return type `bytes`

key_may_match(*key*, *filter*)

Check if the key is maybe in the filter.

Parameters

- **key** (*bytes*) – Key for a single entry inside the database
- **filter** (*bytes*) – Contains the data returned by a preceding call to create_filter on this class

Returns This method must return `True` if the key was in the list of keys passed to create_filter(). This method may return `True` or `False` if the key was not on the list, but it should aim to return `False` with a high probability.

Return type `bool`

name()

Return the name of this policy. Note that if the filter encoding changes in an incompatible way, the name returned by this method must be changed. Otherwise, old incompatible filters may be passed to methods of this type.

Return type `bytes`

SliceTransform

class `rocksdb.interfaces.`**`SliceTransform`**

SliceTransform is currently used to implement the 'prefix-API' of rocksdb. https://github.com/facebook/rocksdb/wiki/Proposal-for-prefix-API

transform(*src*)

Parameters **src** (*bytes*) – Full key to extract the prefix from.

Returns A tuple of two interges `(offset, size)`. Where the first integer is the offset within the `src` and the second the size of the prefix after the offset. Which means the prefix is generted by `src[offset:offset+size]`

Return type `(int, int)`

in_domain(*src*)

Decide if a prefix can be extraced from `src`. Only if this method returns `True` `transform()` will be called.

Parameters **src** (*bytes*) – Full key to check.

Return type `bool`

in_range(*prefix*)

Checks if prefix is a valid prefix

Parameters **prefix** (*bytes*) – Prefix to check.

Returns `True` if `prefix` is a valid prefix.

Return type `bool`

name()

Return the name of this transformation.

Return type `bytes`

1.3.4 Backup and Restore

BackupEngine

class `rocksdb.`**`BackupEngine`**

__init__(*backup_dir*)

Creates a object to manage backup of a single database.

Parameters **backup_dir** (*unicode*) – Where to keep the backup files. Has to be different than db.db_name. For example db.db_name + '/backups'.

`create_backup`(*db*, *flush_before_backup=False*)
Triggers the creation of a backup.

Parameters

- **db** (`rocksdb.DB`) – Database object to backup.
- **flush_before_backup** (*bool*) – If `True` the current memtable is flushed.

`restore_backup`(*backup_id*, *db_dir*, *wal_dir*)
Restores the backup from the given id.

Parameters

- **backup_id** (*int*) – id of the backup to restore.
- **db_dir** (*unicode*) – Target directory to restore backup.
- **wal_dir** (*unicode*) – Target directory to restore backuped WAL files.

`restore_latest_backup`(*db_dir*, *wal_dir*)
Restores the latest backup.

Parameters

- **db_dir** (*unicode*) – see `restore_backup()`
- **wal_dir** (*unicode*) – see `restore_backup()`

`stop_backup`()
Can be called from another thread to stop the current backup process.

`purge_old_backups`(*num_backups_to_keep*)
Deletes all backups (oldest first) until "num_backups_to_keep" are left.

Parameters **num_backups_to_keep** (*int*) – Number of backupfiles to keep.

`delete_backup`(*backup_id*)

Parameters **backup_id** (*int*) – Delete the backup with the given id.

`get_backup_info`()
Returns information about all backups.

It returns a list of dict's where each dict as the following keys.

`backup_id` (int): id of this backup.

`timestamp` (int): Seconds since epoch, when the backup was created.

`size` (int): Size in bytes of the backup.

1.4 Changelog

1.4.1 Upcoming Version

Target is rocksdb 3.0

1.4.2 Version 0.2.1

Make CompactRange available: `rocksdb.DB.compact_range()`

1.4.3 Version 0.2

This version works with RocksDB version 2.8.fb. Now you have access to the more advanced options of rocksdb. Like changing the memtable or SST representation. It is also possible now to enable *Universal Style Compaction*.

- Fixed issue 3. Which fixed the change of prefix_extractor from raw-pointer to smart-pointer.
- Support the new `rocksdb.Options.verify_checksums_in_compaction` option.
- Add `rocksdb.Options.table_factory` option. So you could use the new 'PlainTableFactories' which are optimized for in-memory-databases.
  - https://github.com/facebook/rocksdb/wiki/PlainTable-Format
  - https://github.com/facebook/rocksdb/wiki/How-to-persist-in-memory-RocksDB-database%3F
- Add `rocksdb.Options.memtable_factory` option.
- Add options `rocksdb.Options.compaction_style` and `rocksdb.Options.compaction_options_universa` to change the compaction style.
- Update documentation to the new default values
  - allow_mmap_reads=true
  - allow_mmap_writes=false
  - max_background_flushes=1
  - max_open_files=5000
  - paranoid_checks=true
  - disable_seek_compaction=true
  - level0_stop_writes_trigger=24
  - level0_slowdown_writes_trigger=20
- Document new property names for `rocksdb.DB.get_property()`.

1.4.4 Version 0.1

Initial version. Works with rocksdb version 2.7.fb.

CHAPTER 2

Contributing

Source can be found on github. Feel free to fork and send pull-requests or create issues on the github issue tracker

CHAPTER 3

RoadMap/TODO

No plans so far. Please submit wishes to the github issues.

CHAPTER 4

Indices and tables

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