cPy-LevelDB Documentation

Release 0.4

Fu Haiping

January 24, 2012

CONTENTS

The cPy-LevelDB is a python binding for Google's LevelDB. It is written in pure C and based on LevelDB's c API. The goal is to be super strict for ultimate portability, no dependencies, and generic embeddability.

The binding is still considered alpha but is undergoing active development.

cPy-LevelDB Tutorial An overview of the python API for LevelDB.

Building the LevelDB Python bindings How to build the binding from source code.

LevelDB Object How to use LevelDB object.

Iteration Object How to work with key-value iterator.

Snapshots Object Snapshots provide consistent read-only views over the entire state of the key-value store.

WriteBatch Object The WriteBatch holds a sequence of edits to be made to the database, and these edits within the batch are applied in order.

API Docs Doxygen-generated API docs.

Source code The source code is hosted on GitHub.

CHAPTER

ONE

CPY-LEVELDB TUTORIAL

This document shows how to use LevelDB from Python. If you are not familiar with LevelDB, you will want to get a brief overview of the database and its example API. The official tutorial is a great place to start.

Next, you will want to install and run LevelDB.

A working Python program complete with examples from this tutorial can be found in the examples folder of the source distribution.

1.1 Simple API

When writing programs with Python, you will be using four different entities: connections, iterators, snapshots, and writebatch.

So, for instance, to create a new connection, start by allocating a `LevelDB` object:

```
import leveldb;
db = leveldb.LevelDB("/tmp/test-leveldb")
```

Next, you can do some operations on the `LevelDB` object, such as `Put`, `Get`, `Delete` and so on:

```
db.Put("hello", "world")
print db.Get("hello")
```

Set any optional values, like a block cache size, you can use a specific parameter when allocating a `LevelDB` object:

```
import leveldb;
db = leveldb.LevelDB(filename = "/tmp/test-leveldb", block_cache_size = 4096 * 128)
db.Put("hello", "world")
print db.Get("hello")
```

When you are finished, close the `LevelDB` object:

```
import leveldb;
db = leveldb.LevelDB("/tmp/test-leveldb")
'''
do operations on the leveldb db.
'''
......
db.Close()
```

There are more details, but that is the basic pattern. Keep this in mind as you learn the API and start using the python package.

1.2 Constructing a LevelDB object with other parameters

Let us start by that connects to the database:

```
import leveldb
import datetime
import uuid

db = leveldb.LevelDB(\
    filename = "/tmp/test-leveldb",\
    create_if_missing = True,\
    error_if_exists = True,\
    paranoid_checks = False,\
    write_buffer_size = 4096 * 32,\
    block_size = 128,\
    block_restart_inteval = 100,\
    block_cache_size = 4096 * 128,\
    compression = True)

for i in range(n):
    db.Put(str(i), "this is item %d"%(i))
print("%s: lookup a item from map..."%(datetime.datetime.now()))
print(db.Get("43"))
print("%s: ok"%(datetime.datetime.now()))
```

1.3 Iteration

The following example demonstrates how to print key,value pairs in a database.

```
import leveldb
db = leveldb.LevelDB("/tmp/test-leveldb/")
batch = leveldb.WriteBatch();
for i in xrange(1000):
    batch.Put(str(i), "hello, hello, hello string %s" %i)
db.Write(batch)

print "Creating iterator"
iter = leveldb.Iterator(db);

print "Seek to First."
iter.First()
print "\n"
print "Print iterator's key"
print iter.Key()
print "\n"
print "Print iterator's value"
print iter.Value()
print "\n"

print "Seek to next"
```

```
iter.Next()
print "\n"
print "Print iterator’s key again"
print iter.Key()
print "\n"
print "Print iterator’s value again"
print iter.Value()
print "\n"

print "Seek to Last"
iter.Last()
print "\n"
print "Print iterator’s key again"
print iter.Key()
print "\n"
print "Print iterator’s value again"
print iter.Value()
print "\n"
```

The following example show how to iterate all key-value pairs in a datebase.

```
import leveldb
db = leveldb.LevelDB("/tmp/test-leveldb/")

batch = leveldb.WriteBatch();

for i in xrange(1000):
    batch.Put(str(i), "hello, hello, hello string %s" %i)
db.Write(batch)

print "Creating iterator"

iter = leveldb.Iterator(db);
iter.First()
while True:
    if (iter.Validate()):
        print iter.Key()
        print iter.Value()
        iter.Next()
    else:
        break
```

The following variation shows how to process just the keys in the range [start,limit):

```
import datetime
import uuid
import leveldb

def main():
    n=100
    print("%s: write %d items to db.."%(datetime.datetime.now(),n))
    db = leveldb.LevelDB("./leveldb.db")
    for i in range(n):
        db.Put(str(i), "this is item %d"%(i))

    print("%s: lookup items from map between range key_from to key_to..."\
        %(datetime.datetime.now()))
    print list(db.RangeIter(key_from = '80', key_to = '90'))
    print("%s: ok"%(datetime.datetime.now()))
    return 0

if __name__ == '__main__':
    main()
```

1.4 Synchronous Writes

By default, each write to `LevelDB` is asynchronous: it returns after pushing the write from the process into the operating system. The transfer from operating system memory to the underlying persistent storage happens asynchronously. The `sync` flag can be turned on for a particular write to make the write operation not return until the data being written has been pushed all the way to persistent storage. (On `Posix` systems, this is implemented by calling either `fsync(...)` or `fdatasync(...)` or `msync(..., MS_SYNC)` before the write operation returns.)

Asynchronous writes are often more than a thousand times as fast as synchronous writes. The downside of asynchronous writes is that a crash of the machine may cause the last few updates to be lost. Note that a crash of just the writing process (i.e., not a reboot) will not cause any loss since even when `sync` is false, an update is pushed from the process memory into the operating system before it is considered done.

Asynchronous writes can often be used safely. For example, when loading a large amount of data into the database you can handle lost updates by restarting the bulk load after a crash. A hybrid scheme is also possible where every Nth write is synchronous, and in the event of a crash, the bulk load is restarted just after the last synchronous write finished by the previous run. (The synchronous write can update a marker that describes where to restart on a crash.)

`WriteBatch` provides an alternative to asynchronous writes. Multiple updates may be placed in the same WriteBatch and applied together using a synchronous write The extra cost of the synchronous write will be amortized across all of the writes in the batch.

```
import leveldb
db = leveldb.LevelDB("/tmp/test-leveldb/")
batch = leveldb.WriteBatch();

for i in xrange(1000):
        batch.Put(str(i), "hello, hello, hello string %s" %i)
print "Print a string before WriteBatch takes effect. "
print db.Get("888")
print "\n"
db.Write(batch)
print "Print a string after WriteBatch takes effect. "
print db.Get("888")
```

1.5 Snapshots

Snapshots provide consistent read-only views over the entire state of the key-value store. The following example shows how to work with `Snapshot` object.

```
import leveldb
db = leveldb.LevelDB("/tmp/test-leveldb/")

db.Put("hello", "world")
db.Put("1", "111111")
db.Put("2", "222222")
db.Put("3", "333333")

print "Creating snapshot."
snap = leveldb.Snapshot(db)
print "\n"

print "Deleting key 1."
db.Delete("1")
print "\n"

print "Getting key 1."
print db.Get("1")
print "\n"

print "Applying snapshot."
snap.Set()
print "\n"

print "Getting key 1 again."
print db.Get("1")
print "\n"


print "Resetting snapshot."
snap.Reset()
print "\n"


print "Getting key 1 again."
print db.Get("1")
print "\n"
# Need to release the Snapshot object.
snap.Release()
print "O.K."
```

1.6 Further Reading

This overview just touches on the basics of using LevelDB from Python. For more examples, check out the other documentation pages, and have a look at the package's test cases.

CHAPTER

TWO

BUILDING THE LEVELDB PYTHON BINDINGS

First checkout the version you want to build, *Always build from a particular tag, since HEAD may be a work in progress,* for example, to build version 0.4, run:

```
git checkout v0.4
```

Then follow the build steps below:

First of all, you need to build the included `snappy` and `leveldb` library.

2.1 Building `Snappy`

```
$ cd snappy
$ ./configure && make && make install
```

2.2 Building `LevelDB`

```
$ cd leveldb
$ make

g++ -c -I. -I./include -fno-builtin-memcmp -DLEVELDB_PLATFORM_POSIX -DLEVELDB_CSTDATOMIC_PRESENT\
-std=c++0x -pthread -DOS_LINUX -O2 -DNDEBUG -DSNAPPY util/histogram.cc -o util/histogram.o
... ...
g++ -c -I. -I./include -fno-builtin-memcmp -DLEVELDB_PLATFORM_POSIX -DLEVELDB_CSTDATOMIC_PRESENT\
-std=c++0x -pthread -DOS_LINUX -O2 -DNDEBUG        -DSNAPPY util/options.cc -o util/options.o
rm -f libleveldb.a
ar -rs libleveldb.a ./db/builder.o ./db/c.o ./db/db_impl.o ./db/db_iter.o ./db/filename.o\
... ./util/histogram.o ./util/logging.o ./util/options.o ./util/status.o
ar: creating libleveldb.a
```

2.3 Building `cPy-LevelDB`

```
$ python setup.py build

running build
running build_py
running build_ext
building 'leveldb' extension
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes\
-fPIC -I/usr/include/python2.6 -c src/snapshot.c -o build/temp.linux-i686-2.6/src/snapshot.o\
-Wall -pedantic -I./leveldb/include/ -shared -std=gnu99 -fPIC -g -D_GNU_SOURCE
... ...
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes\
-fPIC -I/usr/include/python2.6 -c src/range_iterator.c -o build/temp.linux-i686-2.6/src/range_iterat
-Wall -pedantic -I./leveldb/include/ -shared -std=gnu99 -fPIC -g -D_GNU_SOURCE
gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions build/temp.linux-i686-2.6/src/initmodule.o\
build/temp.linux-i686-2.6/src/leveldb.o build/temp.linux-i686-2.6/src/write_batch.o\
build/temp.linux-i686-2.6/src/iterator.o build/temp.linux-i686-2.6/src/snapshot.o\
build/temp.linux-i686-2.6/src/range_iterator.o -o build/lib.linux-i686-2.6/leveldb.so\
-L./leveldb -static -lleveldb -lsnappy -lpthread
```

2.4 Dependencies

The binding itself has no dependencies.

CHAPTER

THREE

LEVELDB OBJECT

3.1 Simple Usage

3.2 LevelDB API

LevelDB(*filename, [create_if_missing, error_if_exists, paranoid_checks, write_buffer_size, block_size, max_open_files, block_restart_interval, block_cache_size, compression]*)

Construct a connection to LevelDB datebase.Here are the parameters explaination:

`filename`: Specifing the datebase filename.

`create_if_missing`: If the datebase specified by `filename` does not exists, then create a new datebase.

`error_if_exists`: If the datebase specified by `filename` exits, then error occurs.

`paranoid_checks`: Period checks.

`write_buffer_size`: Specifing the write buffer size.

`block_size`: Specifing the block size.

`max_open_files`: Specifing the max number of files that can be opened.

`block_restart_interval`: Specifing the block restart interval.

`block_cache_size`: Specifing the block cache size.

`compression`: Specifing data compressed or not.

Put(*key, value, [sync]*)

Add a key/value pair to database, with an optional synchronous disk write.

Get(*key, [verify_checksums, fill_cache]*)

Get a value from the database.

Delete(*key, [sync]*)

Delete a value in the database.

Write(*writebach, [sync]*)

Apply a writebatch in database.

Property()

Get a property value.

`RepairDB`(*filename, [create_if_missing, error_if_exists]*)

Repair database.

`RangeIter`(*key_from, key_to, [verify_checksums, fill_cache, include_value]*)

Range iterator.

`GetApproximateSizes`(*num_ranges, range_start_key, range_limit_key*)

Get approximate sizes.

`Close`()

Close database.

CHAPTER

FOUR

ITERATION OBJECT

4.1 Simple Usage

4.2 Iterator API

Iterator(*leveldb*)

Construct an iterator based on the current LevelDB state.

Validate()

Validate whether an iterator is valid, it is useful to verify an iterator exceeding out of bounds.

First()

Seek to *First* key-value pair.

Last()

Seek to *Last* key-value pair.

Seek()

Iterator seek, find the specified `key`.

Next()

Iterator to Next.

Prev()

Iterator to Prev.

Key()

Get key throuth current iterator.

Value()

Get value throuth current iterator.

GetError()

Get iterator error.

Destroy()

Destroy iterator.

CHAPTER

FIVE

SNAPSHOTS OBJECT

Snapshots provide consistent read-only views over the entire state of the key-value store.

5.1 Simple Usage

5.2 Snapshots API

`Snapshot``(`*db*`)`

Construct a snapshot based on the current LevelDB state.

`Set``()`

Set snapshot.

`Reset``()`

Reset snapshot to the current state.

`Release``()`

Release snapshot

WRITEBATCH OBJECT

6.1 Simple Usage

6.2 WriteBatch API

`WriteBatch``()`

Construct a write batch based on the current database state.

`Put``()`

Add a `Put` operation to batch.

`Delete``()`

Add a `Delete` operation to batch.

`Clear``()`

Clear the current batch.

`Release``()`

Release a batch.

INDEX