tempsdb

Release 0.6.5a2

Piotr Maślanka

Oct 17, 2022

CONTENTS:

CHAPTER

ONE

HOW THIS DOES WORK?

Note: This is about fixed length data time series. For the page about time series, see the *proper page*.

Data is stored in so called chunks. A chunk's last page can be actively appended to, or a chunk is immutable.

When there is a request to fetch some data, a chunk is loaded into memory. It will not be automatically unloaded, to do this, you must periodically call `close_chunks()`.

CHAPTER

TWO

USAGE

Start off by instantiating an object

Note that if you specify a *gzip_level* argument in `create_series()`, GZIP compression will be used.

> **Warning:** Note that gzip-compressed series are very slow to read, since every *seek* is conducted from the beginning. indexed-gzip library hung too often. This will be fixed in the future.

Also, any gzip-opened series will raise a warning, since their support is experimental at best.

You can create new databases via

Then you can create and retrieve particular series:

You retrieve their data via Iterators:

Appending the data is done via `append()`. Since time series are allocated in entire pages, so your files will be padded to a page in size. This makes writes quite fast, as in 99.9% cases it is just a memory operation.

2.1 Logging

tempsdb will log when opening and closing series. To prevent this from happening, just call:

CHAPTER

THREE

EXCEPTIONS

The base TempsDB exception is

The exceptions that inherit from it are:

CHAPTER

FOUR

CHUNK

New in version 0.5.

There are two kinds of chunk - a “normal” chunk and “direct” chunk.

The difference is that a normal chunk will preallocate a page ahead, in order for writes to be fast, while direct chunk will write only as much data as is strictly required.

Only “direct” chunks are capable to be gzipped, since one page is preallocated for normal chunk, which would prevent modifications made post-factum to it.

For your convenience the class `Chunk` was also documented, but don’t use it directly:

Data stored in files is little endian.

A way to tell which chunk are we dealing with is to look at it’s extension. Chunks that have:

- no extension - are normal chunks
- *.direct* extension - are direct chunks
- *.gz* extension - are direct and gzipped chunks

4.1 Normal chunk

A file storing a normal chunk consists as follows:

- 4 bytes unsigned int - block size
- **repeated**
  - 8 bytes unsigned long long - timestamp
  - block_size bytes of data

It’s padded to *page_size* with zeros, and four last bytes is the *unsigned long* amount of entries

4.2 Direct chunk

A file storing a direct chunk consists as follows:

- 4 bytes unsigned int - block size
- **repeated**
  - 8 bytes unsigned long long - timestamp
  - block_size bytes of data

Note that a direct chunk will be able to be gzipped. If it's file name ends with .gz, then it's a direct chunk which is gzipped.

CHAPTER

FIVE

VARIABLE LENGTH SERIES

New in version 0.5.

5.1 How does it work?

They work by breaking down your data into smaller pieces and storing them in separate series, prefixing with length.

For each series you specify so-called length profile. It is a list of ints, each representing a block size for next series created. If an entry cannot fit in the already created series, a new one will be created. Note that the last entry of this array will loop forever, so if you for example put a 1024 byte data in a varlen series of length profile [10, 255] there will be a total of 5 normal time series created to accommodate it, with length of:

- 10
- 255
- 255
- 255
- 255

Note that an entry is written to enough series so that it fits. For example, a 8 byte piece of data would be written to only to the first series.

Each entry is also prefixed by it's length, so the actual size of the first series is larger by that. The size of that field is described by an extra parameter called *size_struct*. It represents an unsigned number.

Note that the only valid sizes of *size_struct* are:

- 1 for maximum length of 255
- 2 for maximum length of 65535
- 3 for maximum length of 16777215
- 4 for maximum length of 2147483647

Also note that variable length series live in a different namespace than standard time series, so you can name them the same.

5.2 Accessing them

Use methods `tempsdb.database.Database.create_varlen_series()` and `tempsdb.database.Database.get_varlen_series()` to obtain instances of following class:

CHAPTER

SIX

INTEGRATION WITH SATELLA'S MEMORYPRESSUREMANAGER

This library integrates itself with Satella MemoryPressureManager.

It will close the non-required chunks when remaining in severity 1 each 30 seconds.

To attach a MPM to a database, use `tempsdb.database.Database.register_memory_pressure_manager()`.

Series will automatically inherit the parent database's *MemoryPressureManager*.

This is an append-only embedded time series library written in Cython.

It tries to use mmap for reads and writes, and in general is as zero-copy as possible (ie. the only time data is unserialized is when a particular entry is read). It also uses iterators.

Visit the project GitHub page!

Stored time series with a 8-bit timestamp and a fixed length of data. So no variable encoding for you!

New in version 0.2.

When mmap fails due to memory issues, this falls back to slower fwrite()/fread() implementation. You can also manually select the descriptor-based implementation if you want to.

New in version 0.5.

Experimental support for gzipping time series is added. Note that reading from gzipped files might be slow, as every seek requires reading the file from the beginning.

Warnings will be issued while using gzipped series to remind you of this fact.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search