tempsdb

Release 0.5a2

Piotr Maślanka

Dec 11, 2020

CONTENTS:

CHAPTER ONE

HOW THIS DOES WORK?

Note: This is about fixed length data time series.

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

class `tempsdb.database.`**`Database`**(*unicode path: str*)

A basic TempsDB object.

After you're done with it, please call `close()`.

If you forget to, the destructor will do that instead and emit a warning.

Parameters **`path`** – path to the directory with the database

Raises ***`DoesNotExist`*** – database does not exist, use *create_database*

Variables **`path`** – path to the directory with the database (str)

`close`(*self*) → int

Close this TempsDB database

`close_all_open_series`(*self*) → int

Closes all open series

`create_series`(*self*, *unicode name*, *int block_size*, *unsigned long entries_per_chunk*, *int page_size=4096*, *bool use_descriptor_based_access=False*) → TimeSeries

Create a new series

Parameters

- **`name`** – name of the series
- **`block_size`** – size of the data field
- **`entries_per_chunk`** – entries per chunk file
- **`page_size`** – size of a single page. Default is 4096
- **`use_descriptor_based_access`** – whether to use descriptor based access instead of mmap. Default is False

Returns new series

Raises

- **`ValueError`** – block size was larger than page_size plus a timestamp
- ***`AlreadyExists`*** – series with given name already exists

`create_varlen_series`(*self*, *unicode name*, *list length_profile*, *int size_struct*, *unsigned long entries_per_chunk*) → VarlenSeries

Create a new variable length series

Parameters

- **`name`** – name of the series
- **`length_profile`** – list of lengths of subsequent chunks
- **`size_struct`** – how many bytes will be used to store length? Valid entries are 1, 2 and 4
- **`entries_per_chunk`** – entries per chunk file

Returns new variable length series

Raises ***`AlreadyExists`*** – series with given name already exists

`get_all_series`(*self*) → list

Stream all series available within this database

Returns a list of series names

Return type tp.List[str]

`get_first_entry_for`(*self*, *unicode name*) → unsigned long long

Get first timestamp stored in a particular series without opening it

Parameters **`name`** – series name

Returns first timestamp stored in this series

Raises

- ***`DoesNotExist`*** – series does not exist
- **`ValueError`** – timestamp does not have any data

`get_open_series`(*self*) → list

Return all open series

Returns open series

Return type tp.List[*TimeSeries*]

`get_series`(*self*, *unicode name: str*, *bool use_descriptor_based_access=False*) → TimeSeries

Load and return an existing series

Parameters

- **`name`** – name of the series
- **`use_descriptor_based_access`** – whether to use descriptor based access instead of mmap, default is False

Returns a loaded time series

Raises ***`DoesNotExist`*** – series does not exist

`get_varlen_series`(*self*, *unicode name*) → VarlenSeries

Load and return an existing variable length series

Parameters **`name`** – name of the series

Returns a loaded varlen series

Raises ***`DoesNotExist`*** – series does not exist

`register_memory_pressure_manager`(*self*, *mpm*) → int

Register a satella MemoryPressureManager to close chunks if low on memory.

Parameters **`mpm`** (*`satella.instrumentation.memory.MemoryPressureManager`*) – MemoryPressureManager to use

sync(*self*) → int
Synchronize all the data with the disk

You can create new databases via

`tempsdb.database.`**`create_database`**(*unicode path*) → Database
Creates a new, empty database

Parameters **`path`** – path where the DB directory will be put

Returns a Database object

Raises ***`AlreadyExists`*** – the directory exists

Then you can create and retrieve particular series:

class `tempsdb.series.`**`TimeSeries`**(*unicode path: str, unicode name: str, use_descriptor_based_access: bool = False*)
A single time series. This maps each timestamp (unsigned long long) to a block of data of length block_size.

When you're done with this, please call *`close()`*.

If you forget to, the destructor will do that instead, and a warning will be emitted.

Variables

- **`last_entry_ts`** – timestamp of the last entry added or 0 if no entries yet (int)
- ***`last_entry_synced`*** – timestamp of the last synchronized entry (int)
- **`block_size`** – size of the writable block of data (int)
- **`path`** – path to the directory containing the series (str)
- **`descriptor_based_access`** – are all chunks using descriptor-based access? (bool)
- **`name`** – name of the series (str)
- **`metadata`** – extra data (tp.Optional[dict])

append(*self, unsigned long long timestamp, bytes data*) → int
Append an entry.

Parameters

- **`timestamp`** – timestamp, must be larger than current last_entry_ts
- **`data`** – data to write

Raises

- **`ValueError`** – Timestamp not larger than previous timestamp or invalid block size
- ***`InvalidState`*** – the resource is closed

append_padded(*self, unsigned long long timestamp, bytes data*) → int
Same as *`append()`* but will accept data shorter than block_size.

It will be padded with zeros.

Parameters

- **`timestamp`** – timestamp, must be larger than current last_entry_ts
- **`data`** – data to write

Raises

- **`ValueError`** – Timestamp not larger than previous timestamp or invalid block size

- `InvalidState` – the resource is closed

`close`(*self*) → int

Close the series.

No further operations can be executed on it afterwards.

`close_chunks`(*self*) → int

Close all chunks opened by read requests that are not referred to anymore.

No-op if closed.

`delete`(*self*) → int

Erase this series from the disk. Series must be opened to do that.

Raises `InvalidState` – series is not opened

`disable_mmap`(*self*) → int

Switches to descriptor-based file access method for the entire series, and all chunks open inside.

`enable_mmap`(*self*) → int

Switches to mmap-based file access method for the entire series, and all chunks open inside.

This will try to enable mmap on every chunk, but if mmap fails due to recoverable errors, it will remain in descriptor-based mode.

Raises `Corruption` – mmap failed due to an irrecoverable error

`get_current_value`(*self*) → tuple

Return latest value of this series

Returns tuple of (timestamp, value)

Return type tp.Tuple[int, bytes]

Raises `ValueError` – series has no data

`iterate_range`(*self*, *unsigned long long start*, *unsigned long long stop*) → Iterator

Return an iterator through collected data with given timestamps.

Parameters

- `start` – timestamp to start at
- `stop` – timestamp to stop at

Returns an iterator with the data

Raises `ValueError` – start larger than stop

`mark_synced_up_to`(*self*, *unsigned long long timestamp*) → int

Mark the series as synced up to particular timestamp.

This will additionally sync the metadata.

Parameters `timestamp` – timestamp of the last synced entry

`open_chunks_mmap_size`(*self*) → unsigned long

Calculate how much RAM does the mmaped space take

Returns how much RAM, in bytes, do the opened chunks consume?

`set_metadata`(*self*, *dict new_meta*) → int

Set a new value for the `metadata` property.

This writes the disk.

Parameters `new_meta` – new value of metadata property

sync(*self*) → int

Synchronize the data kept in the memory with these kept on disk

Raises ***InvalidState*** – the resource is closed

trim(*self, unsigned long long timestamp*) → int

Delete all entries earlier than timestamp that are closed.

Note that this will drop entire chunks, so it may be possible that some entries will linger on.

This will affect only closed chunks. Chunks ready to delete that are closed after this will not be deleted, as *trim()* will need to be called again.

Parameters **timestamp** – timestamp to delete entries earlier than

You retrieve their data via Iterators:

class tempsdb.iterators.**Iterator**(*TimeSeries parent: TimeSeries, start: int, stop: int, chunks: tp.List[Chunk]*)

Iterator that allows iterating through result.

Can be used as a context manager:

```
>>> with series.iterate_range(0, 5000) as it:
>>>     for timestamp, value in it:
>>>         ...
```

It will close itself automatically via destructor, if you forget to call close.

At most basic this implements an iterator interface, iterating over tp.Tuple[int, bytes] - timestamp and data

When you're done call *close()* to release the resources.

A warning will be emitted in the case that destructor has to call *close()*.

close(*self*) → int

Close this iterator, release chunks.

It is imperative that you call this, otherwise some chunks might remain in memory.

This is hooked by destructor, but release it manually ASAP.

No-op if iterator is already closed.

next_item(*self*) → tuple

Return next element or None, if list was exhausted

Returns next element

Return type tp.Optional[tp.Tuple[int, bytes]]

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.

CHAPTER THREE

EXCEPTIONS

The base TempsDB exception is

class tempsdb.exceptions.**TempsDBError**
Base class for TempsDB errors

The exceptions that inherit from it are:

class tempsdb.exceptions.**DoesNotExist**
The required resource does not exist

class tempsdb.exceptions.**Corruption**
Corruption was detected in the dataset

class tempsdb.exceptions.**InvalidState**
An attempt was made to write to a resource that's closed

class tempsdb.exceptions.**AlreadyExists**
Provided object already exists

class tempsdb.exceptions.**StillOpen**
This resource has outstanding references and cannot be closed

CHAPTER

FOUR

CHUNK

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

class tempsdb.chunks.**Chunk**(*parent: tp.Optional[TimeSeries], unicode path: str, page_size: int, use_descriptor_access: bool = False*)

Represents a single chunk of time series.

This also implements an iterator interface, and will iterate with tp.Tuple[int, bytes], as well as a sequence protocol.

This will try to mmap opened files, but if mmap fails due to not enough memory this will use descriptor-based access.

Parameters

- **parent** – parent time series
- **path** – path to the chunk file
- **use_descriptor_access** – whether to use descriptor based access instead of mmap

Variables

- **path** – path to the chunk (str)
- **min_ts** – timestamp of the first entry stored (int)
- **max_ts** – timestamp of the last entry stored (int)
- **block_size** – size of the data entries (int)
- **entries** – amount of entries in this chunk (int)
- **page_size** – size of the page (int)

append(*self*, *unsigned long long timestamp*, *bytes data*) → int

Append a record to this chunk.

Might range from very fast (just a memory operation) to quite slow (adding a new page to the file).

Simultaneous writing is not thread-safe.

Timestamp and data is not checked for, this is supposed to be handled by *TimeSeries*.

Parameters

- **timestamp** – timestamp of the entry
- **data** – data to write

Raises ***InvalidState*** – chunk is closed

close(*self*, *bool force=False*) → int

Close the chunk and close the allocated resources

Parameters **`force`** – whether to close the chunk even if it's open somewhere

Raises ***`StillOpen`*** – this chunk has a parent attached and the parent says that this chunk is still being referred to

`delete`(*self*) → int
Close and delete this chunk.

`find_left`(*self*, *unsigned long long timestamp*) → unsigned int
Return an index i of position such that ts[i] <= timestamp and (timestamp-ts[i]) -> min.

Used as bound in searches: you start from this index and finish at *`find_right()`*.

Parameters **`timestamp`** – timestamp to look for, must be smaller or equal to largest element in the chunk

Returns index such that ts[i] <= timestamp and (timestamp-ts[i]) -> min, or length of the array if timestamp is larger than largest element in this chunk

`find_right`(*self*, *unsigned long long timestamp*) → unsigned int
Return an index i of position such that ts[i] > timestamp and (ts[i]-timestamp) -> min

Used as bound in searches: you start from *`find_right()`* and finish at this inclusive.

Parameters **`timestamp`** – timestamp to look for

Returns index such that ts[i] > timestamp and (ts[i]-timestamp) -> min

`get_byte_of_piece`(*self*, *unsigned int index*, *int byte_index*) → int
Return a particular byte of given element at given index.

When index is negative, or larger than block_size, the behaviour is undefined

Parameters

- **`index`** – index of the element
- **`byte_index`** – index of the byte

Returns value of the byte

Raises **`ValueError`** – index too large

`get_mmap_size`(*self*) → unsigned long

Returns how many bytes are mmaped?

Return type int

`get_slice_of_piece_at`(*self*, *unsigned int index*, *int start*, *int stop*) → bytes
Return a slice of data from given element

Parameters

- **`index`** – index of the element
- **`start`** – starting offset of data
- **`stop`** – stopping offset of data

Returns a byte slice

`get_slice_of_piece_starting_at`(*self*, *unsigned int index*, *int start*) → bytes
Return a slice of data from given element starting at given index to the end

Parameters

- **`index`** – index of the element

- **start** – starting index

Returns a byte slice

get_timestamp_at (*self*, *unsigned int index*) → unsigned long long

Return a timestamp at a particular location

Passing an invalid index will result in an undefined behaviour.

Parameters **index** – index of element

Returns the timestamp

get_value_at (*self*, *unsigned int index*) → bytes

Return only the value at a particular index, numbered from 0

Returns value at given index

iterate_indices (*self*, *unsigned long starting_entry*, *unsigned long stopping_entry*)

Return a partial iterator starting at starting_entry and ending at stopping_entry (exclusive).

Parameters

- **starting_entry** – index of starting entry
- **stopping_entry** – index of stopping entry

Returns an iterator

Return type tp.Iterator[tp.Tuple[int, bytes]]

switch_to_descriptor_based_access (*self*) → int

Switch self to descriptor-based access instead of mmap.

No-op if already in descriptor based mode.

switch_to_mmap_based_access (*self*) → int

Switch self to mmap-based access instead of descriptor-based.

No-op if already in mmap mode.

Raises ***Corruption*** – unable to mmap file due to an unrecoverable error

Data stored in files is little endian.

A file storing a 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

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 4294967295

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:

class tempsdb.varlen.**VarlenSeries**(*unicode path: str*, *unicode name: str*)

A time series housing variable length data.

It does that by splitting the data into chunks and encoding them in multiple series.

Parameters

- **path** – path to directory containing the series
- **name** – name of the series

append(*self*, *unsigned long long timestamp*, *bytes data*) → int

Append an entry to the series

Parameters

- **`timestamp`** – timestamp to append it with
- **`data`** – data to write

Raises `ValueError` – too long an entry

`close`(*self*) → int

Close this series.

No-op if already closed.

Raises `StillOpen` – some references are being held

`delete`(*self*) → int

Erases this variable length series from the disk.

Closes this series as a side-effect.

`get_maximum_length`(*self*) → long long

Returns maximum length of an element capable of being stored in this series

`iterate_range`(*self*, *unsigned long long start*, *unsigned long long stop*, *bool direct_bytes=False*) → *VarlenIterator*

Return an iterator with the data

`last_entry_synced`

Returns timestamp of the last entry synchronized. Starting value is 0

`mark_synced_up_to`(*self*, *unsigned long long timestamp*) → int

Mark the series as synchronized up to particular period

Parameters `timestamp` – timestamp of synchronization

`trim`(*self*, *unsigned long long timestamp*) → int

Try to delete all entries younger than timestamp

Parameters `timestamp` – timestamp that separates alive entries from the dead

class `tempsdb.varlen.`**`VarlenIterator`**(*VarlenSeries parent: VarlenSeries*, *start: int*, *stop: int*, *direct_bytes: bool = False*)

A result of a varlen series query.

This iterator will close itself when completed. If you break out of it's iteration, please close it youself via `close()`

If you forget to do that, a warning will be issued and the destructor will close it automatically.

Parameters

- **`parent`** – parent series
- **`start`** – started series
- **`stop`** – stopped series
- **`direct_bytes`** – whether to iterate with bytes values instead of `VarlenEntry`. Note that setting this to True will result in a performance drop, since it will copy, but it should be faster if your typical entry is less than 20 bytes.

`close`(*self*) → int

Close this iterator and release all the resources

No-op if already closed.

`get_next`(*self*) → *VarlenEntry*

Return next element of the iterator, or None if no more available.

class tempsdb.varlen.**VarlenEntry**(*VarlenSeries parent: VarlenSeries, chunks: tp.List[Chunk], item_no: tp.List[int]*)

An object representing the value.

It is preferred for an proxy to exist, instead of copying data. This serves make tempsdb far more zero-copy, but it's worth it only if your values are routinely longer than 20-40 bytes.

This behaves as a bytes object, in particular it can be sliced, iterated, and it's length obtained. It also overloads __bytes__. It's also directly comparable and hashable, and boolable.

This acquires a reference to the chunk it refers, and releases it upon destruction.

Once `to_bytes()` is called, it's result will be cached.

close(*self*) → int

Close this object and release all the references.

It is not necessary to call, since the destructor will call this. .. warning:: Do not let your VarlenEntries outlive the iterator itself!

It will be impossible to close the iterator.

endswith(*self*, *bytes v*) → bool

Check whether this sequence ends with provided bytes.

This will run faster than *bytes(v).endswith(b'test')* since it will fetch only the required amount of bytes.

Parameters **v** – bytes to check

Returns whether the sequence ends with provided bytes

get_byte_at(*self*, *int index*) → int

Return a byte at a particular index

Parameters **index** – index of the byte

Returns the value of the byte

Raises **ValueError** – index too large

length(*self*) → int

Returns self length

slice(*self*, *int start*, *int stop*) → bytes

Returns a slice of the entry

Parameters

- **start** – position to start at
- **stop** – position to stop at

Returns a slice of this entry

Raises **ValueError** – stop was smaller than start or indices were invalid

startswith(*self*, *bytes v*) → bool

Check whether this sequence starts with provided bytes.

This will run faster than *bytes(v).startswith(b'test')* since it will fetch only the required amount of bytes.

Parameters **v** – bytes to check

Returns whether the sequence starts with provided bytes

timestamp(*self*) → unsigned long long

Returns timestamp assigned to this entry `to_bytes`(*self*) → bytes

Returns value as bytes

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.

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.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX