GitDB Documentation

Release 0.5.3

Sebastian Thiel

May 28, 2017

Contents

Contents:

CHAPTER 1

Overview

The *GitDB* project implements interfaces to allow read and write access to git repositories. In its core lies the *db* package, which contains all database types necessary to read a complete git repository. These are the `LooseObjectDB`, the `PackedDB` and the `ReferenceDB` which are combined into the `GitDB` to combine every aspect of the git database.

For this to work, GitDB implements pack reading, as well as loose object reading and writing. Data is always encapsulated in streams, which allows huge files to be handled as well as small ones, usually only chunks of the stream are kept in memory for processing, never the whole stream at once.

Interfaces are used to describe the API, making it easy to provide alternate implementations.

Installing GitDB

Its easiest to install gitdb using the *pip* program:

```
$ pip install gitdb
```

As the command will install gitdb in your respective python distribution, you will most likely need root permissions to authorize the required changes.

If you have downloaded the source archive, the package can be installed by running the `setup.py` script:

```
$ python setup.py install
```

Getting Started

It is advised to have a look at the *Usage Guide* for a brief introduction on the different database implementations.

Source Repository

The latest source can be cloned using git from github:

- https://github.com/gitpython-developers/gitdb

License Information

GitDB is licensed under the New BSD License.

CHAPTER 2

Usage Guide

This text briefly introduces you to the basic design decisions and accompanying types.

Design

The *GitDB* project models a standard git object database and implements it in pure python. This means that data, being classified by one of four types, can can be stored in the database and will in future be referred to by the generated SHA1 key, which is a 20 byte string within python.

GitDB implements *RW* access to loose objects, as well as *RO* access to packed objects. Compound Databases allow to combine multiple object databases into one.

All data is read and written using streams, which effectively prevents more than a chunk of the data being kept in memory at once mostly[1].

Streams

In order to assure the object database can handle objects of any size, a stream interface is used for data retrieval as well as to fill data into the database.

Basic Stream Types

There are two fundamentally different types of streams, **IStream**s and **OStream**s. IStreams are mutable and are used to provide data streams to the database to create new objects.

OStreams are immutable and are used to read data from the database. The base of this type, **OInfo**, contains only type and size information of the queried object, but no stream, which is slightly faster to retrieve depending on the database.

[1] When reading streams from packs, all deltas are currently applied and the result written into a memory map before the first byte is returned. Future versions of the delta-apply algorithm might improve on this.

OStreams are tuples, IStreams are lists. Both, OInfo and OStream, have the same member ordering which allows quick conversion from one type to another.

Data Query and Data Addition

Databases support query and/or addition of objects using simple interfaces. They are called **ObjectDBR** for read-only access, and **ObjectDBW** for write access to create new objects.

Both have two sets of methods, one of which allows interacting with single objects, the other one allowing to handle a stream of objects simultaneously and asynchronously.

Acquiring information about an object from a database is easy if you have a SHA1 to refer to the object:

```
ldb = LooseObjectDB(fixture_path("../../../.git/objects"))

for sha1 in ldb.sha_iter():
    oinfo = ldb.info(sha1)
    ostream = ldb.stream(sha1)
    assert oinfo[:3] == ostream[:3]

    assert len(ostream.read()) == ostream.size
# END for each sha in database
```

To store information, you prepare an *IStream* object with the required information. The provided stream will be read and converted into an object, and the respective 20 byte SHA1 identifier is stored in the IStream object:

```
data = "my data"
istream = IStream("blob", len(data), StringIO(data))

# the object does not yet have a sha
assert istream.binsha is None
ldb.store(istream)
# now the sha is set
assert len(istream.binsha) == 20
assert ldb.has_object(istream.binsha)
```

Asynchronous Operation

For each read or write method that allows a single-object to be handled, an *_async* version exists which reads items to be processed from a channel, and writes the operation's result into an output channel that is read by the caller or by other async methods, to support chaining.

Using asynchronous operations is easy, but chaining multiple operations together to form a complex one would require you to read the docs of the *async* package. At the current time, due to the *GIL*, the *GitDB* can only achieve true concurrency during zlib compression and decompression if big objects, if the respective c modules where compiled in *async*.

Asynchronous operations are scheduled by a *ThreadPool* which resides in the *gitdb.util* module:

```
from gitdb.util import pool

# set the pool to use two threads
pool.set_size(2)
```

```
# synchronize the mode of operation
pool.set_size(0)
```

Use async methods with readers, which supply items to be processed. The result is given through readers as well:

```
from async import IteratorReader

# Create a reader from an iterator
reader = IteratorReader(ldb.sha_iter())

# get reader for object streams
info_reader = ldb.stream_async(reader)

# read one
info = info_reader.read(1)[0]

# read all the rest until depletion
ostreams = info_reader.read()
```

Databases

A database implements different interfaces, one if which will always be the *ObjectDBR* interface to support reading of object information and streams.

The *Loose Object Database* as well as the *Packed Object Database* are *File Databases*, hence they operate on a directory which contains files they can read.

File databases implementing the *ObjectDBW* interface can also be forced to write their output into the specified stream, using the `set_ostream` method. This effectively allows you to redirect its output to anywhere you like.

Compound Databases are not implementing their own access type, but instead combine multiple database implementations into one. Examples for this database type are the *Reference Database*, which reads object locations from a file, and the *GitDB* which combines loose, packed and referenced objects into one database interface.

For more information about the individual database types, please see the *API Reference*, and the unittests for the respective types.

CHAPTER 3

API Reference

Database.Base

Contains implementations of database retrieveing objects

class gitdb.db.base.**ObjectDBR**
Defines an interface for object database lookup. Objects are identified either by their 20 byte bin sha

has_object(*sha*)

Returns True if the object identified by the given 20 bytes binary sha is contained in the database

info(*sha*)

Returns OInfo instance

Parameters **sha** – bytes binary sha

Raises **BadObject** –

sha_iter()
Return iterator yielding 20 byte shas for all objects in this data base

size()

Returns amount of objects in this database

stream(*sha*)

Returns OStream instance

Parameters **sha** – 20 bytes binary sha

Raises **BadObject** –

class gitdb.db.base.**ObjectDBW**(**args*, ***kwargs*)
Defines an interface to create objects in the database

ostream()

Returns overridden output stream this instance will write to, or None if it will write to the default stream

set_ostream(*stream*)
Adjusts the stream to which all data should be sent when storing new objects

Parameters **stream** – if not None, the stream to use, if None the default stream will be used.

Returns previously installed stream, or None if there was no override

Raises **TypeError** – if the stream doesn't have the supported functionality

store(*istream*)
Create a new object in the database :return: the input istream object with its sha set to its corresponding value

Parameters **istream** – IStream compatible instance. If its sha is already set to a value, the object will just be stored in the our database format, in which case the input stream is expected to be in object format (header + contents).

Raises **IOError** – if data could not be written

class gitdb.db.base.**FileDBBase**(*root_path*)
Provides basic facilities to retrieve files of interest, including caching facilities to help mapping hexsha's to objects

db_path(*rela_path*)

Returns the given relative path relative to our database root, allowing to pontentially access datafiles

root_path()

Returns path at which this db operates

class gitdb.db.base.**CompoundDB**
A database which delegates calls to sub-databases.

Databases are stored in the lazy-loaded _dbs attribute. Define _set_cache_ to update it with your databases

databases()

Returns tuple of database instances we use for lookups

has_object(*sha*)

info(*sha*)

partial_to_complete_sha_hex(*partial_hexsha*)

Returns 20 byte binary sha1 from the given less-than-40 byte hexsha (bytes or str)

Parameters **partial_hexsha** – hexsha with less than 40 byte

Raises **AmbiguousObjectName** –

sha_iter()

size()

Returns total size of all contained databases

stream(*sha*)

update_cache(*force=False*)

class gitdb.db.base.**CachingDB**
A database which uses caches to speed-up access

update_cache(*force=False*)
Call this method if the underlying data changed to trigger an update of the internal caching structures.

Parameters **force** – if True, the update must be performed. Otherwise the implementation may decide not to perform an update if it thinks nothing has changed.

Returns True if an update was performed as something change indeed

Database.Git

class gitdb.db.git.**GitDB**(*root_path*)
A git-style object database, which contains all objects in the 'objects' subdirectory

IMPORTANT: The usage of this implementation is highly discouraged as it fails to release file-handles. This can be a problem with long-running processes and/or big repositories.

LooseDBCls
alias of LooseObjectDB

PackDBCls
alias of PackedDB

ReferenceDBCls
alias of ReferenceDB

alternates_dir = 'info/alternates'

loose_dir = ''

ostream()

packs_dir = 'pack'

set_ostream(*ostream*)

store(*istream*)

Database.Loose

class gitdb.db.loose.**LooseObjectDB**(*root_path*)
A database which operates on loose object files

has_object(*sha*)

info(*sha*)

new_objects_mode = 292

object_path(*hexsha*)

Returns path at which the object with the given hexsha would be stored, relative to the database root

partial_to_complete_sha_hex(*partial_hexsha*)

Returns 20 byte binary sha1 string which matches the given name uniquely

Parameters **name** – hexadecimal partial name (bytes or ascii string)

Raises

- **AmbiguousObjectName** –

- **BadObject** –

readable_db_object_path(*hexsha*)

Returns readable object path to the object identified by hexsha

Raises **BadObject** – If the object file does not exist

set_ostream(*stream*)

Raises **TypeError** – if the stream does not support the Sha1Writer interface

sha_iter()

size()

store(*istream*)
note: The sha we produce will be hex by nature

stream(*sha*)

stream_chunk_size = 4096000

Database.Memory

Contains the MemoryDatabase implementation

class gitdb.db.mem.**MemoryDB**
A memory database stores everything to memory, providing fast IO and object retrieval. It should be used to buffer results and obtain SHAs before writing it to the actual physical storage, as it allows to query whether object already exists in the target storage before introducing actual IO

has_object(*sha*)

info(*sha*)

set_ostream(*stream*)

sha_iter()

size()

store(*istream*)

stream(*sha*)

stream_copy(*sha_iter*, *odb*)
Copy the streams as identified by sha's yielded by sha_iter into the given odb The streams will be copied directly **Note:** the object will only be written if it did not exist in the target db :return: amount of streams actually copied into odb. If smaller than the amount

of input shas, one or more objects did already exist in odb

Database.Pack

Module containing a database to deal with packs

class gitdb.db.pack.**PackedDB**(*root_path*)
A database operating on a set of object packs

entities()

Returns list of pack entities operated upon by this database

has_object(*sha*)

info(*sha*)

partial_to_complete_sha(*partial_binsha*, *canonical_length*)

Returns 20 byte sha as inferred by the given partial binary sha

Parameters

- **partial_binsha** – binary sha with less than 20 bytes
- **canonical_length** – length of the corresponding canonical representation. It is required as binary sha's cannot display whether the original hex sha had an odd or even number of characters

Raises

- **AmbiguousObjectName** –
- **BadObject** –

sha_iter()

size()

store(*istream*)

Storing individual objects is not feasible as a pack is designed to hold multiple objects. Writing or rewriting packs for single objects is inefficient

stream(*sha*)

update_cache(*force=False*)

Update our cache with the acutally existing packs on disk. Add new ones, and remove deleted ones. We keep the unchanged ones

Parameters **force** – If True, the cache will be updated even though the directory does not appear to have changed according to its modification timestamp.

Returns True if the packs have been updated so there is new information, False if there was no change to the pack database

Database.Reference

class gitdb.db.ref.**ReferenceDB**(*ref_file*)

A database consisting of database referred to in a file

ObjectDBCls = None

update_cache(*force=False*)

Base

Module with basic data structures - they are designed to be lightweight and fast

class gitdb.base.**OInfo**(**args*)

Carries information about an object in an ODB, providing information about the binary sha of the object, the type_string as well as the uncompressed size in bytes.

It can be accessed using tuple notation and using attribute access notation:

```
assert dbi[0] == dbi.binsha
assert dbi[1] == dbi.type
assert dbi[2] == dbi.size
```

The type is designed to be as lightweight as possible.

binsha

Returns our sha as binary, 20 bytes

hexsha

Returns our sha, hex encoded, 40 bytes

size

type

type_id

class gitdb.base.**OPackInfo**(**args*)
As OInfo, but provides a type_id property to retrieve the numerical type id, and does not include a sha.

Additionally, the pack_offset is the absolute offset into the packfile at which all object information is located. The data_offset property points to the absolute location in the pack at which that actual data stream can be found.

pack_offset

size

type

type_id

class gitdb.base.**ODeltaPackInfo**(**args*)
Adds delta specific information, Either the 20 byte sha which points to some object in the database, or the negative offset from the pack_offset, so that pack_offset - delta_info yields the pack offset of the base object

delta_info

class gitdb.base.**OStream**(**args*, ***kwargs*)
Base for object streams retrieved from the database, providing additional information about the stream. Generally, ODB streams are read-only as objects are immutable

read(*size=-1*)

stream

class gitdb.base.**OPackStream**(**args*)
Next to pack object information, a stream outputting an undeltified base object is provided

read(*size=-1*)

stream

class gitdb.base.**ODeltaPackStream**(**args*)
Provides a stream outputting the uncompressed offset delta information

read(*size=-1*)

stream

class gitdb.base.**IStream**(*type*, *size*, *stream*, *sha=None*)
Represents an input content stream to be fed into the ODB. It is mutable to allow the ODB to record information about the operations outcome right in this instance.

It provides interfaces for the OStream and a StreamReader to allow the instance to blend in without prior conversion.

The only method your content stream must support is 'read'

binsha

error

Returns the error that occurred when processing the stream, or None

hexsha

Returns our sha, hex encoded, 40 bytes

read(*size=-1*)
Implements a simple stream reader interface, passing the read call on to our internal stream

size

stream

type

class gitdb.base.**InvalidOInfo**(*sha*, *exc*)
Carries information about a sha identifying an object which is invalid in the queried database. The exception attribute provides more information about the cause of the issue

binsha

error

Returns exception instance explaining the failure

hexsha

class gitdb.base.**InvalidOStream**(*sha*, *exc*)
Carries information about an invalid ODB stream

Functions

Contains basic c-functions which usually contain performance critical code Keeping this code separate from the beginning makes it easier to out-source it into c later, if required

gitdb.fun.**is_loose_object**(*m*)

Returns True the file contained in memory map m appears to be a loose object. Only the first two bytes are needed

gitdb.fun.**loose_object_header_info**(*m*)

Returns tuple(type_string, uncompressed_size_in_bytes) the type string of the object as well as its uncompressed size in bytes.

Parameters **m** – memory map from which to read the compressed object data

gitdb.fun.**msb_size**(*data*, *offset=0*)

Returns tuple(read_bytes, size) read the msb size from the given random access data starting at the given byte offset

gitdb.fun.**pack_object_header_info**(*data*)

Returns tuple(type_id, uncompressed_size_in_bytes, byte_offset) The type_id should be interpreted according to the `type_id_to_type_map` map The byte-offset specifies the start of the actual zlib compressed datastream

Parameters **m** – random-access memory, like a string or memory map

gitdb.fun.**write_object**(*type*, *size*, *read*, *write*, *chunk_size=4096000*)

Write the object as identified by type, size and source_stream into the target_stream

Parameters

- **type** – type string of the object
- **size** – amount of bytes to write from source_stream
- **read** – read method of a stream providing the content data
- **write** – write method of the output stream
- **close_target_stream** – if True, the target stream will be closed when the routine exits, even if an error is thrown

Returns The actual amount of bytes written to stream, which includes the header and a trailing newline

gitdb.fun.**loose_object_header**(*type*, *size*)

Returns bytes representing the loose object header, which is immediately followed by the content stream of size 'size'

gitdb.fun.**stream_copy**(*read*, *write*, *size*, *chunk_size*)

Copy a stream up to size bytes using the provided read and write methods, in chunks of chunk_size

Note: its much like stream_copy utility, but operates just using methods

gitdb.fun.**apply_delta_data**(*src_buf*, *src_buf_size*, *delta_buf*, *delta_buf_size*, *write*)

Apply data from a delta buffer using a source buffer to the target file

Parameters

- **src_buf** – random access data from which the delta was created
- **src_buf_size** – size of the source buffer in bytes
- **delta_buf_size** – size fo the delta buffer in bytes
- **delta_buf** – random access delta data
- **write** – write method taking a chunk of bytes

Note: transcribed to python from the similar routine in patch-delta.c

gitdb.fun.**is_equal_canonical_sha**(*canonical_length*, *match*, *sha1*)

Returns True if the given lhs and rhs 20 byte binary shas The comparison will take the canonical_length of the match sha into account, hence the comparison will only use the last 4 bytes for uneven canonical representations

Parameters

- **match** – less than 20 byte sha
- **sha1** – 20 byte sha

gitdb.fun.**connect_deltas**(*dstreams*)

Read the condensed delta chunk information from dstream and merge its information into a list of existing delta chunks

Parameters **dstreams** – iterable of delta stream objects, the delta to be applied last comes first, then all its ancestors in order

Returns DeltaChunkList, containing all operations to apply

class gitdb.fun.**DeltaChunkList**

List with special functionality to deal with DeltaChunks. There are two types of lists we represent. The one was created bottom-up, working towards the latest delta, the other kind was created top-down, working from the latest delta down to the earliest ancestor. This attribute is queryable after all processing with is_reversed.

apply(*bbuf*, *write*)

Only used by public clients, internally we only use the global routines for performance

check_integrity(*target_size=-1*)

Verify the list has non-overlapping chunks only, and the total size matches target_size :param target_size: if not -1, the total size of the chain must be target_size :raise AssertionError: if the size doen't match

compress()

Alter the list to reduce the amount of nodes. Currently we concatenate add-chunks :return: self

lbound()

Returns leftmost byte at which this chunklist starts

rbound()

Returns rightmost extend in bytes, absolute

size()

Returns size of bytes as measured by our delta chunks

gitdb.fun.**create_pack_object_header**(*obj_type*, *obj_size*)

Returns string defining the pack header comprised of the object type and its incompressed size in bytes

Parameters

- **obj_type** – pack type_id of the object
- **obj_size** – uncompressed size in bytes of the following object stream

Pack

Contains PackIndexFile and PackFile implementations

class gitdb.pack.**PackIndexFile**(*indexpath*)

A pack index provides offsets into the corresponding pack, allowing to find locations for offsets faster.

close()

index_v2_signature = '\xfftOc'

index_version_default = 2

indexfile_checksum()

Returns 20 byte sha representing the sha1 hash of this index file

offsets()

Returns sequence of all offsets in the order in which they were written

Note: return value can be random accessed, but may be immmutable

packfile_checksum()

Returns 20 byte sha representing the sha1 hash of the pack file

partial_sha_to_index(*partial_bin_sha*, *canonical_length*)

Returns index as in *sha_to_index* or None if the sha was not found in this index file

Parameters

- **partial_bin_sha** – an at least two bytes of a partial binary sha as bytes
- **canonical_length** – length of the original hexadecimal representation of the given partial binary sha

Raises **AmbiguousObjectName** –

path()

Returns path to the packindexfile

sha_to_index(*sha*)

Returns index usable with the `offset` or `entry` method, or None if the sha was not found in this pack index

Parameters **sha** – 20 byte sha to lookup

size()

Returns amount of objects referred to by this index

version()

class gitdb.pack.**PackFile**(*packpath*)

A pack is a file written according to the Version 2 for git packs

As we currently use memory maps, it could be assumed that the maximum size of packs therefor is 32 bit on 32 bit systems. On 64 bit systems, this should be fine though.

Note: at some point, this might be implemented using streams as well, or streams are an alternate path in the case memory maps cannot be created for some reason - one clearly doesn't want to read 10GB at once in that case

checksum()

Returns 20 byte sha1 hash on all object sha's contained in this file

close()

collect_streams(*offset*)

Returns list of pack streams which are required to build the object at the given offset. The first entry of the list is the object at offset, the last one is either a full object, or a REF_Delta stream. The latter type needs its reference object to be locked up in an ODB to form a valid delta chain. If the object at offset is no delta, the size of the list is 1.

Parameters **offset** – specifies the first byte of the object within this pack

data()

Returns read-only data of this pack. It provides random access and usually is a memory map.

Note This method is unsafe as it returns a window into a file which might be larger than than the actual window size

first_object_offset = 12

footer_size = 20

info(*offset*)
Retrieve information about the object at the given file-absolute offset

Parameters **offset** – byte offset

Returns OPackInfo instance, the actual type differs depending on the type_id attribute

pack_signature = 1346454347

pack_version_default = 2

path()

Returns path to the packfile

size()

Returns The amount of objects stored in this pack

stream(*offset*)
Retrieve an object at the given file-relative offset as stream along with its information

Parameters **offset** – byte offset

Returns OPackStream instance, the actual type differs depending on the type_id attribute

stream_iter(*start_offset=0*)

Returns iterator yielding OPackStream compatible instances, allowing to access the data in the pack directly.

Parameters **start_offset** – offset to the first object to iterate. If 0, iteration starts at the very first object in the pack.

Note: Iterating a pack directly is costly as the datastream has to be decompressed to determine the bounds between the objects

version()

Returns the version of this pack

class gitdb.pack.**PackEntity**(*pack_or_index_path*)
Combines the PackIndexFile and the PackFile into one, allowing the actual objects to be resolved and iterated

IndexFileCls
alias of *PackIndexFile*

PackFileCls
alias of *PackFile*

close()

collect_streams(*sha*)
As `PackFile.collect_streams`, but takes a sha instead of an offset. Additionally, ref_delta streams will be resolved within this pack. If this is not possible, the stream will be left alone, hence it is adivsed to check for unresolved ref-deltas and resolve them before attempting to construct a delta stream.

Parameters **sha** – 20 byte sha1 specifying the object whose related streams you want to collect

Returns list of streams, first being the actual object delta, the last being a possibly unresolved base object.

Raises **BadObject** –

collect_streams_at_offset(*offset*)
As the version in the PackFile, but can resolve REF deltas within this pack For more info, see collect_streams

Parameters **offset** – offset into the pack file at which the object can be found

classmethod **create**(*object_iter*, *base_dir*, *object_count=None*, *zlib_compression=1*)
Create a new on-disk entity comprised of a properly named pack file and a properly named and corresponding index file. The pack contains all OStream objects contained in object iter. :param base_dir: directory which is to contain the files :return: PackEntity instance initialized with the new pack

Note: for more information on the other parameters see the write_pack method

index()

Returns the underlying pack index file instance

info(*sha*)
Retrieve information about the object identified by the given sha

Parameters **sha** – 20 byte sha1

Raises **BadObject** –

Returns OInfo instance, with 20 byte sha

info_at_index(*index*)
As info, but uses a PackIndexFile compatible index to refer to the object

info_iter()

Returns Iterator over all objects in this pack. The iterator yields OInfo instances

is_valid_stream(*sha*, *use_crc=False*)
Verify that the stream at the given sha is valid.

Parameters

- **use_crc** – if True, the index' crc is run over the compressed stream of the object, which is much faster than checking the sha1. It is also more prone to unnoticed corruption or manipulation.
- **sha** – 20 byte sha1 of the object whose stream to verify whether the compressed stream of the object is valid. If it is a delta, this only verifies that the delta's data is valid, not the data of the actual undeltified object, as it depends on more than just this stream. If False, the object will be decompressed and the sha generated. It must match the given sha

Returns True if the stream is valid

Raises

- **UnsupportedOperation** – If the index is version 1 only
- **BadObject** – sha was not found

pack()

Returns the underlying pack file instance

stream(*sha*)
Retrieve an object stream along with its information as identified by the given sha

Parameters **sha** – 20 byte sha1

Raises **BadObject** –

Returns OStream instance, with 20 byte sha

stream_at_index(*index*)
As `stream`, but uses a PackIndexFile compatible index to refer to the object

stream_iter()

Returns iterator over all objects in this pack. The iterator yields OStream instances

classmethod write_pack(*object_iter*, *pack_write*, *index_write=None*, *object_count=None*, *zlib_compression=1*)
Create a new pack by putting all objects obtained by the object_iterator into a pack which is written using the pack_write method. The respective index is produced as well if index_write is not Non.

Parameters

- **object_iter** – iterator yielding odb output objects
- **pack_write** – function to receive strings to write into the pack stream
- **indx_write** – if not None, the function writes the index file corresponding to the pack.
- **object_count** – if you can provide the amount of objects in your iteration, this would be the place to put it. Otherwise we have to pre-iterate and store all items into a list to get the number, which uses more memory than necessary.
- **zlib_compression** – the zlib compression level to use

Returns tuple(pack_sha, index_binsha) binary sha over all the contents of the pack and over all contents of the index. If index_write was None, index_binsha will be None

Note: The destination of the write functions is up to the user. It could be a socket, or a file for instance

Note: writes only undeltified objects

Streams

class `gitdb.stream.`**DecompressMemMapReader**(*m*, *close_on_deletion*, *size=None*)
Reads data in chunks from a memory map and decompresses it. The client sees only the uncompressed data, respective file-like read calls are handling on-demand buffered decompression accordingly

A constraint on the total size of bytes is activated, simulating a logical file within a possibly larger physical memory area

To read efficiently, you clearly don't want to read individual bytes, instead, read a few kilobytes at least.

Note: The chunk-size should be carefully selected as it will involve quite a bit of string copying due to the way the zlib is implemented. Its very wasteful, hence we try to find a good tradeoff between allocation time and number of times we actually allocate. An own zlib implementation would be good here to better support streamed reading - it would only need to keep the mmap and decompress it into chunks, that's all ...

close()
Close our underlying stream of compressed bytes if this was allowed during initialization :return: True if we closed the underlying stream :note: can be called safely

compressed_bytes_read()

Returns number of compressed bytes read. This includes the bytes it took to decompress the header (if there was one)

data()

Returns random access compatible data we are working on

max_read_size = 524288

classmethod new(*m*, *close_on_deletion=False*)

Create a new DecompressMemMapReader instance for acting as a read-only stream This method parses the object header from m and returns the parsed type and size, as well as the created stream instance.

Parameters

- **m** – memory map on which to operate. It must be object data (header + contents)
- **close_on_deletion** – if True, the memory map will be closed once we are being deleted

read(*size=-1*)

seek(*offset*, *whence=0*)

Allows to reset the stream to restart reading :raise ValueError: If offset and whence are not 0

class gitdb.stream.**FDCompressedSha1Writer**(*fd*)

Digests data written to it, making the sha available, then compress the data and write it to the file descriptor

Note: operates on raw file descriptors **Note:** for this to work, you have to use the close-method of this instance

close()

exc = IOError('Failed to write all bytes to filedescriptor',)

fd

sha1

write(*data*)

Raises **IOError** – If not all bytes could be written

Returns length of incoming data

zip

class gitdb.stream.**DeltaApplyReader**(*stream_list*)

A reader which dynamically applies pack deltas to a base object, keeping the memory demands to a minimum.

The size of the final object is only obtainable once all deltas have been applied, unless it is retrieved from a pack index.

The uncompressed Delta has the following layout (MSB being a most significant bit encoded dynamic size):

•MSB Source Size - the size of the base against which the delta was created

•MSB Target Size - the size of the resulting data after the delta was applied

•A list of one byte commands (cmd) which are followed by a specific protocol:

•cmd & 0x80 - copy delta_data[offset:offset+size]

•Followed by an encoded offset into the delta data

•Followed by an encoded size of the chunk to copy

•cmd & 0x7f - insert

- insert cmd bytes from the delta buffer into the output stream
- cmd == 0 - invalid operation (or error in delta stream)

k_max_memory_move = 250000000

classmethod new(*stream_list*)
Convert the given list of streams into a stream which resolves deltas when reading from it.

Parameters **stream_list** – two or more stream objects, first stream is a Delta to the object that you want to resolve, followed by N additional delta streams. The list's last stream must be a non-delta stream.

Returns Non-Delta OPackStream object whose stream can be used to obtain the decompressed resolved data

Raises **ValueError** – if the stream list cannot be handled

read(*count=0*)

seek(*offset*, *whence=0*)
Allows to reset the stream to restart reading

Raises **ValueError** – If offset and whence are not 0

size

Returns number of uncompressed bytes in the stream

type

type_id

class gitdb.stream.**Sha1Writer**
Simple stream writer which produces a sha whenever you like as it degests everything it is supposed to write

sha(*as_hex=False*)

Returns sha so far

Parameters **as_hex** – if True, sha will be hex-encoded, binary otherwise

sha1

write(*data*)

Raises **IOError** – If not all bytes could be written

Parameters **data** – byte object

Returns length of incoming data

class gitdb.stream.**FlexibleSha1Writer**(*writer*)
Writer producing a sha1 while passing on the written bytes to the given write function

write(*data*)

writer

class gitdb.stream.**ZippedStoreShaWriter**
Remembers everything someone writes to it and generates a sha

buf

close()

getvalue()

Returns string value from the current stream position to the end

seek(*offset*, *whence=0*)
Seeking currently only supports to rewind written data Multiple writes are not supported

write(*data*)

zip

class gitdb.stream.**FDCompressedSha1Writer**(*fd*)
Digests data written to it, making the sha available, then compress the data and write it to the file descriptor

Note: operates on raw file descriptors **Note:** for this to work, you have to use the close-method of this instance

close()

exc = IOError('Failed to write all bytes to filedescriptor',)

fd

sha1

write(*data*)

Raises **IOError** – If not all bytes could be written

Returns length of incoming data

zip

class gitdb.stream.**FDStream**(*fd*)
A simple wrapper providing the most basic functions on a file descriptor with the fileobject interface. Cannot use os.fdopen as the resulting stream takes ownership

close()

fileno()

read(*count=0*)

tell()

write(*data*)

class gitdb.stream.**NullStream**
A stream that does nothing but providing a stream interface. Use it like /dev/null

close()

read(*size=0*)

write(*data*)

Types

Module containing information about types known to the database

Utilities

class gitdb.util.**LazyMixin**
Base class providing an interface to lazily retrieve attribute values upon first access. If slots are used, memory

will only be reserved once the attribute is actually accessed and retrieved the first time. All future accesses will return the cached value as stored in the Instance's dict or slot.

class gitdb.util.**LockedFD**(*filepath*)

This class facilitates a safe read and write operation to a file on disk. If we write to 'file', we obtain a lock file at 'file.lock' and write to that instead. If we succeed, the lock file will be renamed to overwrite the original file.

When reading, we obtain a lock file, but to prevent other writers from succeeding while we are reading the file.

This type handles error correctly in that it will assure a consistent state on destruction.

note with this setup, parallel reading is not possible

commit()

When done writing, call this function to commit your changes into the actual file. The file descriptor will be closed, and the lockfile handled.

Note can be called multiple times

open(*write=False*, *stream=False*)

Open the file descriptor for reading or writing, both in binary mode.

Parameters

- **write** – if True, the file descriptor will be opened for writing. Other wise it will be opened read-only.
- **stream** – if True, the file descriptor will be wrapped into a simple stream object which supports only reading or writing

Returns fd to read from or write to. It is still maintained by this instance and must not be closed directly

Raises

- **IOError** – if the lock could not be retrieved
- **OSError** – If the actual file could not be opened for reading

note must only be called once

rollback()

Abort your operation without any changes. The file descriptor will be closed, and the lock released.

Note can be called multiple times

gitdb.util.**allocate_memory**(*size*)

Returns a file-protocol accessible memory block of the given size

gitdb.util.**byte_ord**(*b*)

Return the integer representation of the byte string. This supports Python 3 byte arrays as well as standard strings.

gitdb.util.**file_contents_ro**(*fd*, *stream=False*, *allow_mmap=True*)

Returns read-only contents of the file represented by the file descriptor fd

Parameters

- **fd** – file descriptor opened for reading
- **stream** – if False, random access is provided, otherwise the stream interface is provided.
- **allow_mmap** – if True, its allowed to map the contents into memory, which allows large files to be handled and accessed efficiently. The file-descriptor will change its position if this is False

gitdb.util.**file_contents_ro_filepath**(*filepath*, *stream=False*, *allow_mmap=True*, *flags=0*)
Get the file contents at filepath as fast as possible

Returns random access compatible memory of the given filepath

Parameters

- **stream** – see file_contents_ro
- **allow_mmap** – see file_contents_ro
- **flags** – additional flags to pass to os.open

Raises **OSError** – If the file could not be opened

Note for now we don't try to use O_NOATIME directly as the right value needs to be shared per database in fact. It only makes a real difference for loose object databases anyway, and they use it with the help of the flags parameter

gitdb.util.**make_sha**(*source=''*)
A python2.4 workaround for the sha/hashlib module fiasco

Note From the dulwich project

gitdb.util.**remove**(**args*, ***kwargs*)

gitdb.util.**sliding_ro_buffer**(*filepath*, *flags=0*)

Returns a buffer compatible object which uses our mapped memory manager internally ready to read the whole given filepath

gitdb.util.**to_bin_sha**(*sha*)

gitdb.util.**to_hex_sha**(*sha*)

Returns hexified version of sha

CHAPTER 4

Discussion of Algorithms

Introduction

As you know, the pure-python object database support for GitPython is provided by the GitDB project. It is meant to be my backup plan to ensure that the DataVault (http://www.youtube.com/user/ByronBates99?feature=mhum#p/c/2A5C6EF5BDA8DB5C) can handle reading huge files, especially those which were consolidated into packs. A nearly fully packed state is anticipated for the data-vaults repository, and reading these packs efficiently is an essential task.

This document informs you about my findings in the struggle to improve the way packs are read to reduce memory usage required to handle huge files. It will try to conclude where future development could go to assure big delta-packed files can be read without the need of 8GB+ RAM.

GitDB's main feature is the use of streams, hence the amount of memory used to read a database object is minimized, at the cost of some additional processing overhead to keep track of the stream state. This works great for legacy objects, which are essentially a zip-compressed byte-stream.

Streaming data from delta-packed objects is far more difficult though, and only technically possible within certain limits, and at relatively high processing costs. My first observation was that there doesn't appear to be 'the one and only' algorithm which is generally superior. They all have their pros and cons, but fortunately this allows the implementation to choose the one most suited based on the amount of delta streams, as well as the size of the base, which allows an early and cheap estimate of the target size of the final data.

Traditional Delta-Apply-Algorithms

The brute-force CGit delta-apply algorithm

CGit employs a simple and relatively brute-force algorithm, which resolves all delta streams recursively. When the recursion reaches the base level of the deltas, it will be decompressed into a buffer, then the first delta gets decompressed into a second buffer. From that, the target size of the delta can be extracted, to allocated a third buffer to hold the result of the operation, which consists of reading the delta stream byte-wise, to apply the operations in order, as described by

single-byte opcodes. During recursion, each target buffer of the preceding delta-apply operation is used as base buffer for the next delta-apply operation, until the last delta was applied, leaving the final target buffer as result.

About Delta-Opcodes

There are only two kinds of opcodes, 'add-bytes' and 'copy-from-base'. One 'add-bytes' opcode can encode up to 7 bit of additional bytes to be copied from the delta stream into the target buffer. A 'copy-from-base' opcode encodes a 32 bit offset into the base buffer, as well as the amount of bytes to be copied, which are up to 2^24 bytes. We may conclude that delta-bases may not be larger than 2^32+2^24 bytes in the current, extensible, implementation. When generating the delta, git prefers copy operations over add operations, as they are much more efficient. Usually, the most recent, or biggest version of a file is used as base, whereas older and smaller versions of the file are expressed by copying only portions of the newest file. As it is not efficiently possible to represent all changes that way, add-bytes operations fill the gap where needed. All this explains why git can only add 128 bytes with one opcode, as it tries to minimize their use. This implies that recent file history can usually be extracted faster than old history, which may involve many more deltas.

Performance considerations

The performance bottleneck of this algorithm appear to be the throughput of your RAM, as both opcodes will just trigger memcpy operations from one memory location to another, times the amount of deltas to apply. This in fact is very fast, even for big files above 100 MB. Memory allocation could become an issue as you need the base buffer, the target buffer as well as the decompressed delta stream in memory at the same time. The continuous allocation and deallocation of possibly big buffers may support memory fragmentation. Whether it really kicks in, especially on 64 bit machines, is unproven though. Nonetheless, the cgit implementation is currently the fastest one.

The brute-force GitDB algorithm

Despite of working essentially the same way as the CGit brute-force algorithm, GitDB minimizes the amount of allocations to 2 + num of deltas. The amount memory allocated peaks whiles the deltas are applied, as the base and target buffer, as well as the decompressed stream, are held in memory. To achieve this, GitDB retrieves all delta-streams in advance, and peaks into their header information to determine the maximum size of the target buffer, just by reading 512 bytes of the compressed stream. If there is more than one delta to apply, the base buffer is set large enough to hold the biggest target buffer required by the delta streams. Now it is possible to iterate all deltas, oldest first, newest last, and apply them using the buffers. At the end of each iteration, the buffers are swapped.

Performance Results

The performance test is performed on an aggressively packed repository with the history of cgit. 5000 sha's are extracted and read one after another. The delta-chains have a length of 0 to about 35. The pure-python implementation can stream the data of all objects (totaling 62,2 MiB) with an average rate of 8.1 MiB/s, which equals about 654 streams/s. There are two bottlenecks: The major is the collection of the delta streams, which involves plenty of pack-lookup. This lookup is expensive in python, and is overly expensive. Its not overly critical though, as it only limits the amount of streams per second, not the actual data rate when applying the deltas. Applying the deltas happens to be the second bottleneck, if the files to be processed get bigger. The more opcodes have to be processed, the more python slow function calls will dominate the result. As an example, it takes nearly 8 seconds to unpack a 125 MB file, where cgit only takes 2.4 s.

To eliminate a few performance concerns, some key routines were rewritten in C. This changes the numbers of this particular test significantly, but not drastically, as the major bottleneck (delta collection) is still in place. Another performance reduction is due to the fact that plenty of other code related to the deltas is still pure-python. Now all 5000 objects can be read at a rate of 11.1 MiB /s, or 892 streams/s. Fortunately, unpacking a big object is now done in 2.5s, which is just a tad slower than cgit, but with possibly less memory fragmentation.

Paving the way towards delta streaming

GitDB's reverse delta aggregation algorithm

The idea of this algorithm is to merge all delta streams into one, which can then be applied in just one go.

In the current implementation, delta streams are parsed into DeltaChunks (->**DC**). Each DC represents one copy-from-base operation, or one or multiple consecutive add-bytes operations. DeltaChunks know about their target offset in the target buffer, and their size. Their target offsets are consecutive, i.e. one chunk ends where the next one begins, regarding their logical extend in the target buffer. Add-bytes DCs additional store their data to apply, copy-from-base DCs store the offset into the base buffer from which to copy bytes.

During processing, one starts with the latest (i.e. topmost) delta stream (->**TDS**), and iterates through its ancestor delta streams (->ADS) to merge them into the growing toplevel delta stream..

The merging works by following a set of rules:

- Merge into the top-level delta from the youngest ancestor delta to the oldest one
- When merging one ADS, iterate from the first to the last chunk in TDS, then:
- skip all add-bytes DCs. If bytes are added, these will always overwrite any operation coming from any ADS at the same offset.
- copy-from-base DCs will copy a slice of the respective portion of the ADS (as defined by their base offset) and use it to replace the original chunk. This acts as a 'virtual' copy-from-base operation.
- Finish the merge once all ADS have been handled, or once the TDS only consists of add-byte DCs. The remaining copy-from-base DCs will copy from the original base buffer accordingly.

Applying the TDS is as straightforward as applying any other DS. The base buffer is required to be kept in memory. In the current implementation, a full-size target buffer is allocated to hold the result of applying the chunk information. Here it is already possible to stream the result, which is feasible only if the memory of the base buffer + the memory of the TDS are smaller than a full size target buffer. Streaming will always make sense if the peak resulting from having the base, target and TDS buffers in memory together is unaffordable.

The memory consumption during the TDS processing is only the condensed delta-bytes, for each ADS an additional index is required which costs 8 byte per DC. When applying the TDS, one requires an allocated base buffer too.The target buffer can be allocated, but may be a writer as well.

Performance Results

The benchmarking context was the same as for the brute-force GitDB algorithm. This implementation is far more complex than the said brute-force implementation, which clearly reflects in the numbers. It's pure-python throughput is at only 1.1 MiB/s, which equals 89 streams/s. The biggest performance bottleneck is the slicing of the parsed delta streams, where the program spends most of its time due to hundred thousands of calls.

To get a more usable version of the algorithm, it was implemented in C, such that python must do no more than two calls to get all the work done. The first prepares the TDS, the second applies it, writing it into a target buffer. The throughput reaches 15.2 MiB/s, which equals 1221 streams/s, which makes it nearly 14 times faster than the pure python version, and amazingly even 1.35 times faster than the brute-force C implementation. As a comparison, cgit is able to stream about 20 MiB when controlling it through a pipe. GitDBs performance may still improve once pack access is reimplemented in C as well.

A 125 MB file took 2.5 seconds to unpack for instance, which is only 20% slower than the c implementation of the brute-force algorithm.

Future work

Another very promising option is that streaming of delta data is indeed possible. Depending on the configuration of the copy-from-base operations, different optimizations could be applied to reduce the amount of memory required for the final processed delta stream. Some configurations may even allow it to stream data from the base buffer, instead of pre-loading it for random access.

The ability to stream files at reduced memory costs would only be feasible for big files, and would have to be paid with extra pre-processing time.

A very first and simple implementation could avoid memory peaks by streaming the TDS in conjunction with a base buffer, instead of writing everything into a fully allocated target buffer.

CHAPTER 5

Changelog

0.6.1

- Fixed possibly critical error, see https://github.com/gitpython-developers/GitPython/issues/220
  - However, it only seems to occur on high-entropy data and didn't reoccour after the fix

0.6.0

- Added support got python 3.X
- Removed all *async* dependencies and all **_async* versions of methods with it.

0.5.4

- Adjusted implementation to use the SlidingMemoryManager by default in python 2.6 for efficiency reasons. In Python 2.4, the StaticMemoryManager will be used instead.

0.5.3

- Added support for smmap. SmartMMap allows resources to be managed and controlled. This brings the implementation closer to the way git handles memory maps, such that unused cached memory maps will automatically be freed once a resource limit is hit. The memory limit on 32 bit systems remains though as a sliding mmap implementation is not used for performance reasons.

0.5.2

- Improved performance of the c implementation, which now uses reverse-delta-aggregation to make a memory bound operation CPU bound.

0.5.1

- Restored most basic python 2.4 compatibility, such that gitdb can be imported within python 2.4, pack access cannot work though. This at least allows Super-Projects to provide their own workarounds, or use everything but pack support.

0.5.0

Initial Release

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z