GitDB Documentation

Release 0.5.3

Sebastian Thiel

November 19, 2014

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.

1.1 Installing GitDB

Its easiest to install gitdb using the *easy_install* program, which is part of the setuptools:

```
$ easy_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
```

1.2 Getting Started

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

1.3 Source Repository

The latest source can be cloned using git from github:

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

1.3.1 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.

2.1 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].

2.2 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.

2.2.1 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.

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

[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.

2.3 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)
```

2.4 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()
```

2.5 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

CHAPTER 4

Discussion of Algorithms

4.1 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.

4.2 Traditional Delta-Apply-Algorithms

4.2.1 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.

4.2.2 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.

4.3 Paving the way towards delta streaming

4.3.1 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.

4.3.2 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 payed 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

5.1 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.

5.2 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.

5.3 0.5.2

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

5.4 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.

5.5 0.5.0

Initial Release

CHAPTER 6

Indices and tables

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