ZS Documentation

Release 0.10.0

Nathaniel J. Smith

Sep 27, 2017

Contents

ZS is a simple, read-only, binary file format designed for distributing, querying, and archiving arbitarily large data sets (up to tens of terabytes and beyond) – so long as those data sets can be represented as a set of arbitrary binary records. Of course it works on small data sets too. You can think of it as an alternative to storing data in tab- or comma-separated files – each line in such a file becomes a record in a ZS file. But ZS has a number of advantages over these traditional formats:

- ZS files are **small**: ZS files (optionally) store data in compressed form. The 3-gram counts from the 2012 US English release of the Google N-grams are distributed as a set of gzipped text files in tab-separated format, and take 1.3 terabytes of space. Uncompressed, this data set comes to more than 10 terabytes (and would be even more if loaded into a database). The same data in a ZS file with the default settings (LZMA compression) takes just 0.75 terabytes – this is more than 41% smaller than the current distribution format, and 13.5x smaller than the raw data.

- Nonetheless, ZS files are **fast**: Decompression is an inherently slow and serial operation, which means that reading compressed files can easily become the bottleneck in an analysis. Google distributes the 3-gram counts in many separate `.gz` files; one of these, for example, contains just the n-grams that begin with the letters "th". Using a single core on a handy compute server, we find that we can get decompressed data out of this `.gz` file at ~190 MB/s. At this rate, reading this one file takes more than 47 minutes – and that's before we even begin analyzing the data inside it.

  The LZMA compression used in our ZS file is, on its own, slower than gzip. If we restrict ourselves to a single core, then we can only read our ZS file at ~50 MB/s. However, ZS files allow for multithreaded decompression. Using 8 cores, gunzip runs at... still ~190 MB/s, because gzip decompression cannot be parallelized. On those same 8 cores, our ZS file decompresses at ~390 MB/s – a nearly linear speedup. This is also ~3x faster than our test server can read an *un*compressed file from disk.

- In fact, ZS files are **really, REALLY fast**: Suppose we want to know how many different Google-scanned books published in the USA in 1955 used the phrase "this is fun". ZS files have a limited indexing ability that lets you quickly locate any arbitrary span of records that fall within a given sorted range, or share a certain textual prefix. This isn't as nice as a full-fledged database system that can query on any column, but it can be extremely useful for data sets where the first column (or first several columns) are usually used for lookup. Using our example file, finding the "this is fun" entry takes 5 disk seeks and ~25 milliseconds of CPU time – something like 85 ms all told. (And hot cache performance – e.g., when performing repeated queries in the same file – is even better.) The answer, by the way, is 27 books:

  ```
  $ zs dump --prefix='this is fun\t1955\t' google-books-eng-us-all-20120701-3gram.zs
  this is fun     1955    27      27
  ```

  When this data is stored as gzipped text, then only way to locate an individual record, or span of similar records, is start decompressing the file from the beginning and wait until the records we want happen to scroll by, which in this case – as noted above – could take more than 45 minutes. Using ZS makes this query ~33,000x faster.

- ZS files contain **rich metadata**: In addition to the raw data records, every ZS file contains a set of structured metadata in the form of an arbitrary JSON document. You can use this to store information about this file's record format (e.g., column names), notes on data collection or preprocessing steps, recommended citation information, or whatever you like, and be confident that it will follow your data where-ever it goes.

- ZS files are **network friendly**: Suppose you know you just want to look up a few individual records that are buried inside that 0.75 terabyte file, or want a large span of records that are still much smaller than the full file (e.g., all 3-grams that begin "this is"). With ZS, you don't have to actually download the full 0.75 terabytes of data. Given a URL to the file, the ZS tools can find and fetch just the parts of the file you need, using nothing but standard HTTP. Of course going back and forth to the server does add overhead; if you need to make a large number of queries then it might be faster (and kinder to whoever's hosting the file!) to just download it. But there's no point in throwing around gigabytes of data to answer a kilobyte question.

  If you have the ZS tools installed, you can try it right now. Here's a real trace of a computer in Dallas searching the 3-gram database stored at UC San Diego. Note that the computer in San Diego has no special software installed at all – this is just a static file that's available for download over HTTP:

```
$ time zs dump --prefix='this is fun\t' http://bolete.ucsd.edu/njsmith/google-
↪books-eng-us-all-20120701-3gram.zs
this is fun       1729    1       1
this is fun       1848    1       1
...
this is fun       2008    435     420
this is fun       2009    365     352

Real time elapsed: 1.425 seconds
```

- ZS files are **ever-vigilant**: Computer hardware is simply not reliable, especially on scales of years and terabytes. I've dealt with RAID cards that would occasionally flip a single bit in the data that was being read from disk. How confident are you that this won't be a key bit that totally changes your results? Standard text files provide no mechanism for detecting data corruption. Gzip and other traditional compression formats provide some protection, but it's only guaranteed to work if you read the entire file from start to finish and then remember to check the error code at the end, every time. But ZS is different: it protects every bit of data with 64-bit CRC checksums, and the software we distribute will never show you any data that hasn't first been double-checked for correctness. (Fortunately, the cost of this checking is negligible; all the times quoted above include these checks). If it matters to you whether your analysis gets the right answer, then ZS is a good choice.
- Relying on the ZS format creates **minimal risk**: The ZS file format is simple and *fully documented*; an average programmer with access to standard libraries could write a working decompressor in a few hours. The reference implementation is BSD-licensed, undergoes exhaustive automated testing (>98% coverage) after every checkin, and just in case there are any ambiguities in the English spec, we also have a complete *file format validator*, so you can confirm that your files match the spec and be confident that they will be readable by any compliant implementation.
- ZS files have a name **composed entirely of sibilants**: How many file formats can say *that*?

This manual documents the reference implementation of the ZS file format, which includes both a command-line `zs` tool for manipulating ZS files and a fast and featureful Python API, and also provides a complete specification of the ZS file format in enough detail to allow independent implementations.

Contents:

CHAPTER 1

Project logistics

Documentation: http://zs.readthedocs.org/

Installation: You need either Python **2.7**, or else Python **3.3 or greater**.

Because `zs` includes a C extension, you'll also need a C compiler and Python headers. On Ubuntu or Debian, for example, you get these with:

```
sudo apt-get install build-essential python-dev
```

Once you have the ability to build C extensions, then on Python 3 you should be able to just run:

```
pip install zs
```

On Python 2.7, things are slightly more complicated: here, `zs` requires the `backports.lzma` package, which in turn requires the liblzma library. On Ubuntu or Debian, for example, something like this should work:

```
sudo apt-get install liblzma-dev
pip install backports.lzma
pip install zs
```

`zs` also requires the following packages: `six`, `docopt`, `requests`. However, these are all pure-Python packages which pip will install for you automatically when you run `pip install zs`.

Downloads: http://pypi.python.org/pypi/zs/

Code and bug tracker: https://github.com/njsmith/zs

Contact: Nathaniel J. Smith <nathaniel.smith@ed.ac.uk>

Developer dependencies (only needed for hacking on source):

- Cython: needed to build from checkout
- nose: needed to run tests
- nose-cov: because we use multiprocessing, we need this package to get useful test coverage information
- nginx: needed to run HTTP tests

License: 2-clause BSD, see LICENSE.txt for details.

CHAPTER 2

The command-line `zs` tool

The `zs` tool can be used from the command-line to create, view, and check ZS files.

The main `zs` command on its own isn't very useful. It can tell you what version you have – these docs were built with:

```
$ zs --version
0.10.0
```

And it can tell you what subcommands are available:

```
$ zs --help
ZS: a space-efficient file format format for distributing, archiving,
and querying large data sets.

Usage:
  zs <subcommand> [<args>...]
  zs --version
  zs --help

Available subcommands:
  zs dump      Get contents of a .zs file.
  zs info      Get general metadata about a .zs file.
  zs validate  Check a .zs file for validity.
  zs make      Create a new .zs file with specified contents.

For details, use 'zs <subcommand> --help'.
```

These subcommands are documented further below.

Note: In case you have the Python *zs* package installed, but somehow do not have the `zs` executable available on your path, then it can also be invoked as `python -m zs`. E.g., these two commands do the same thing:

```
$ zs dump myfile.zs
$ python -m zs dump myfile.zs
```

---

zs make

`zs make` allows you to create ZS files. In its simplest form, it just reads in a text file, and writes out a ZS file, treating each line as a separate record.

For example, if we have this data file (a tiny excerpt from the Web 1T dataset released by Google; note that the last whitespace in each line is a tab character):

```
$ cat tiny-4grams.txt
not done explicitly .         42
not done extensive research        225
not done extensive testing        749
not done extensive tests         87
not done extremely well         41
not done fairly .         61
not done fast ,         52
not done fast enough         71
```

Then we can compress it into a ZS file by running:

```
$ zs make '{"corpus": "doc-example"}' tiny-4grams.txt tiny-4grams.zs --codec deflate
zs: Opening new ZS file: tiny-4grams.zs
zs: Reading input file: tiny-4grams.txt
zs: Blocks written: 1
zs: Blocks written: 2
zs: Updating header...
zs: Done.
```

The first argument specifies some arbitrary metadata that will be saved into the ZS file, in the form of a JSON string; the second argument names the file we want to convert; and the third argument names the file we want to create.

The `--codec` argument lets us choose which compression method we use; usually you should stick with the default (which is lzma), but until readthedocs.org responds to our bug report we can't use lzma here in the docs. Sorry.

Note: You must ensure that your file is sorted before running `zs make`. (If you don't, then it will error out and scold you.) GNU sort is very useful for this task – but don't forget to set `LC_ALL=C` in your environment before calling sort, to make sure that it uses ASCIIbetical ordering instead of something locale-specific.

When your file is too large to fit into RAM, GNU sort will spill the data onto disk in temporary files. When your file is too large to fit onto disk, then a useful incantation is:

```
gunzip -c myfile.gz | env LC_ALL=C sort --compress-program=lzop \
   | zs make "{...}" - myfile.zs
```

The `--compress-program` option tells sort to automatically compress and decompress the temporary files using the `lzop` utility, so that you never end up with uncompressed data on disk. (`gzip` also works, but will be slower.)

Many other options are also available:

```
$ zs make --help
Create a new .zs file.

Usage:
  zs make <metadata> <input_file> <new_zs_file>
  zs make [--terminator TERMINATOR | --length-prefixed=TYPE]
          [-j PARALLELISM]
          [--no-spinner]
```

```
          [--branching-factor=FACTOR]
          [--approx-block-size=SIZE]
          [--codec=CODEC] [-z COMPRESS-LEVEL]
          [--no-default-metadata]
          [--]
          <metadata> <input_file> <new_zs_file>
  zs make --help

Arguments:

  <metadata>      Arbitrary JSON-encoded metadata that will be stored in your
                  new ZS file. This must be a JSON "object", i.e., the
                  outermost characters have to be {}. If you're just messing
                  about, then you can just use "{}" here and be done, but for
                  any file that will live for long then we strongly recommend
                  adding more details about what this file is. See the
                  "Metadata conventions" section of the ZS manual for more
                  information.

  <input_file>    A file containing the records to be packed into the
                  new .zs file. Use "-" for stdin. Records must already be
                  sorted in ASCIIbetical order. You may want to do something
                  like:
                    cat myfile.txt | env LC_ALL=C sort | zs make - myfile.zs

  <new_zs_file>  The file to create. Conventionally uses the file extension
                 ".zs".

Input file options:
  --terminator=TERMINATOR    Treat the input file as containing a series of
                             records separated by TERMINATOR. Standard Python
                             string escapes are supported (e.g., "\x00" for
                             NUL-terminated records). The default is
                             appropriate for standard Unix/OS X text files. If
                             your have a text file with Windows-style line
                             endings, then you'll want to use "\r\n"
                             instead. [default: \n]
  --length-prefixed=TYPE     Treat the input file as containing a series of
                             records containing arbitrary binary data, each
                             prefixed by its length in bytes, with this length
                             encoded according to TYPE. (Valid options:
                             uleb128, u64le.)

Processing options:
  -j PARALLELISM             The number of CPUs to use for compression.
                             [default: guess]
  --no-spinner               Disable the progress meter.

Output file options:
  --branching-factor=FACTOR  Number of keys in each *index* block.
                             [default: 1024]
  --approx-block-size=SIZE   Approximate *uncompressed* size of the records in
                             each *data* block, in bytes. [default: 393216]
  --codec=CODEC              Compression algorithm. (Valid options: none,
                             deflate, lzma.) [default: lzma]
  -z COMPRESS-LEVEL, --compress-level=COMPRESS-LEVEL
                             Degree of compression to use. Interpretation
                             depends on the codec in use:
```

```
                              deflate: An integer between 1 and 9.
                                (Default: 6)
                              lzma: One of the strings 0, 0e, 1, or 1e.
                                The number (0 versus 1) indicates the history
                                size used in the compression -- there's no
                                point in using 1 or 1e unless you also
                                increase --approx-block-size. The presence of
                                the "e" turns on "extreme" mode, which is
                                several times slower, but may produce
                                substantially smaller files. (Default: 0e)
  --no-default-metadata       By default, 'zs make' adds an extra "build-info"
                              key to the metadata, recording the time, host,
                              user who created the file, and zs library
                              version. This option disables this behaviour.
```

zs info

`zs info` displays some general information about a ZS file. For example:

```
$ zs info tiny-4grams.zs
{
    "root_index_offset": 380,
    "root_index_length": 41,
    "total_file_length": 421,
    "codec": "deflate",
    "data_sha256": "403b706aa1f8f5d1d2ffd2765507239bd5a5025bde3f89df8035f8a5b9348b11",
    "metadata": {
        "build-info": {
            "time": "2014-04-29T12:41:59.660529Z",
            "version": "zs 0.9.0",
            "user": "njs",
            "host": "branna.vorpus.org"
        },
        "corpus": "doc-example"
    },
    "statistics": {
        "root_index_level": 1
    }
}
```

The most interesting part of this output might be the `"metadata"` field, which contains arbitrary metadata describing the file. Here we see that our custom key was indeed added, and that `zs make` also added some default metadata. (If we wanted to suppress this we could have used the `--no-default-metadata` option.) The `"data_sha256"` field is, as you might expect, a SHA-256 hash of the data contained in this file – two ZS files will have the same value here if and only if they contain exactly the same logical records, regardless of compression and other details of physical file layout. The `"codec"` field tells us which kind of compression was used. The other fields have to do with more obscure technical aspects of the ZS file format; see the documentation for the *ZS* class and the *file format specification* for details.

`zs info` is fast, even on arbitrarily large files, because it looks at only the header and the root index; it doesn't have to uncompress the actual data. If you find a large ZS file on the web and want to see its metadata before downloading it, you can pass an HTTP URL to `zs info` directly on the command line, and it will download only as much of the file as it needs to.

`zs info` doesn't take many options:

```
$ zs info --help
Display general information from a .zs file's header.

Usage:
  zs info [--metadata-only] [--] <zs_file>
  zs info --help

Arguments:
  <zs_file>  Path or URL pointing to a .zs file. An argument beginning with
             the four characters "http" will be treated as a URL.

Options:
  -m, --metadata-only   Output only the file's metadata, not any general
                        information about it.

Output will be valid JSON.
```

zs dump

So `zs info` tells us *about* the contents of a ZS file, but how do we get our data back out? That's the job of `zs dump`. In the simplest case, it simply dumps the whole file to standard output, with one record per line – the inverse of `zs make`. For example, this lets us "uncompress" our ZS file to recover the original file:

```
$ zs dump tiny-4grams.zs
not done explicitly .         42
not done extensive research        225
not done extensive testing        749
not done extensive tests         87
not done extremely well         41
not done fairly .         61
not done fast ,         52
not done fast enough         71
```

But we can also extract just a subset of the data. For example, we can pull out a single line (notice the use of `\t` to specify a tab character – Python-style backslash character sequences are fully supported):

```
$ zs dump tiny-4grams.zs --prefix="not done extensive testing\t"
not done extensive testing        749
```

Or a set of related ngrams:

```
$ zs dump tiny-4grams.zs --prefix="not done extensive "
not done extensive research        225
not done extensive testing        749
not done extensive tests         87
```

Or any arbitrary range:

```
$ zs dump tiny-4grams.zs --start="not done ext" --stop="not done fast"
not done extensive research        225
not done extensive testing        749
not done extensive tests         87
not done extremely well         41
not done fairly .         61
```

Just like `zs info`, `zs dump` is fast – it reads only the data it needs to to satisfy your query. (Of course, if you request the whole file, then it will read the whole file – but it does this in an optimized way; see the `-j` option if you want to tune how many CPUs it uses for decompression.) And just like `zs info`, `zs dump` can directly take an HTTP URL on the command line, and will download only as much data as it has to.

We also have several options to let us control the output format. ZS files allow records to contain arbitrary data, which means that it's possible to have a record that contains a newline embedded in it. So we might prefer to use some other character to mark the ends of records, like NUL:

```
$ zs dump tiny-4grams.zs --terminator="\x00"
```

...but putting the output from that into these docs would be hard to read. Instead we'll demonstrate with something sillier:

```
$ zs dump tiny-4grams.zs --terminator="XYZZY" --prefix="not done extensive "
not done extensive research        225XYZZYnot done extensive␣
↪testing        749XYZZYnot done extensive tests        87XYZZY
```

Of course, this will still have a problem if any of our records contained the string "XYZZY" – in fact, our records could in theory contain *anything* we might choose to use as a terminator, so if we have an arbitrary ZS file whose contents we know nothing about, then none of the options we've seen so far is guaranteed to work. The safest approach is to instead use a format in which each record is explicitly prefixed by its length. `zs dump` can produce length-prefixed output with lengths encoded in either u64le or uleb128 format (see *Integer representations* for details about what these are).

```
$ zs dump tiny-4grams.zs --prefix="not done extensive " --length-prefixed=u64le | hd
/bin/sh: 1: hd: not found
Exception ignored in: <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>
BrokenPipeError: [Errno 32] Broken pipe
```

Obviously this is mostly intended for when you want to read the data into another program. For example, if you had a ZS file that was compressed using the lzma codec and you wanted to convert it to the deflate codec, the easiest and safest way to do that is with a command like:

```
$ zs dump --length-prefixed=uleb128 myfile-lzma.zs | \
  zs make --length-prefixed=uleb128 --codec=deflate \
      "$(zs info -m myfile-lzma.zs)" - myfile-deflate.zs
```

If you're using Python, of course, the most convenient way to read a ZS file into your program is not to use `zs dump` at all, but to use the *zs* library API directly.

Full options:

```
$ zs dump --help
Unpack some or all of the contents of a .zs file.

Usage:
  zs dump <zs_file>
  zs dump [--start=START] [--stop=STOP] [--prefix=PREFIX]
          [--terminator=TERMINATOR | --length-prefixed=TYPE]
          [-j PARALLELISM]
          [-o FILE]
          [--] <zs_file>
  zs dump --help

Arguments:
  <zs_file>  Path or URL pointing to a .zs file. An argument beginning with
             the four characters "http" will be treated as a URL.
```

```
Selection options:
  --start=START            Output only records which are >= START.
  --stop=STOP              Do not output any records which are >= STOP.
  --prefix=PREFIX          Output only records which begin with PREFIX.

  Python string escapes (e.g., "\n", "\x00") are allowed. All comparisons
  are performed using ASCIIbetical ordering.

Processing options:
  -j PARALLELISM           The number of CPUs to use for decompression. Note
                           that if you know that you are only reading a small
                           number of records, then -j0 may be the fastest
                           option, since it reduces startup overhead.
                           [default: guess]

Output options:
  -o FILE, --output=FILE   Output to the given file, or "-" for stdout.
                           [default: -]

Record framing options:
  --terminator=TERMINATOR  String used to terminate records in output. Python
                           string escapes are allowed (e.g., "\n", "\x00").
                           [default: \n]
  --length-prefixed=TYPE   Instead of terminating records with a marker,
                           prefix each record with its length, encoded as
                           TYPE. (Options: uleb128, u64le)

  ZS files are organized as a collection of records, which may contain
  arbitrary data. By default, these are output as individual lines. However,
  this may not be a great idea if you have records which themselves contain
  newline characters. As an alternative, you can request that they instead be
  terminated by some arbitrary string, or else request that each record be
  prefixed by its length, encoded in either unsigned little-endian base-128
  (uleb128) format or unsigned little-endian 64-bit (u64le) format.
```

Warning: Due to limitations in the multiprocessing module in Python 2, `zs dump` can be poorly behaved if you hit control-C (e.g., refusing to exit).

On a Unix-like platform, if you have a `zs dump` that is ignoring control-C, then try hitting control-Z and then running `kill %zs`.

The easy workaround to this problem is to use Python 3 to run `zs`. The not so easy workaround is to implement a custom process pool manager for Python 2 – patches accepted!

zs validate

This command can be used to fully validate a ZS file for self-consistency and compliance with the specification (see *On-disk layout of ZS files*); this makes it rather useful to anyone trying to write new software to generate ZS files.

It is also useful because it verifies the SHA-256 checksum and all of the per-block checksums, providing extremely strong protection against errors caused by disk failures, cosmic rays, and other such annoyances. However, this is not usually necessary, since the `zs` commands and the *zs* library interface never return any data unless it passes a 64-bit checksum. With ZS you can be sure that your results have not been corrupted by hardware errors, even if you never

run `zs validate` at all.

Full options:

```
$ zs validate --help
Check a .zs file for errors or data corruption.

Usage:
  zs validate [-j PARALLELISM] [--] <zs_file>

Arguments:
  <zs_file>  Path or URL pointing to a .zs file. An argument beginning with
             the four characters "http" will be treated as a URL.

Options:
  -j PARALLELISM             The number of CPUs to use for decompression.
                             [default: guess]
```

CHAPTER 3

The zs library for Python

Quickstart

Using the example file we created when demonstrating *zs make*, we can write:

```
In [1]: from zs import ZS

In [2]: z = ZS("example/tiny-4grams.zs")

In [3]: for record in z:
   ...:     print(record.decode("utf-8"))
   ...:
not done explicitly .	42
not done extensive research	225
not done extensive testing	749
not done extensive tests	87
not done extremely well	41
not done fairly .	61
not done fast ,	52
not done fast enough	71

# Notice that on Python 3.x, we search using byte strings, and we get
# byte strings back.
# (On Python 2.x, byte strings are the same as regular strings.)
In [4]: for record in z.search(prefix=b"not done extensive testing\t"):
   ...:     print(record.decode("utf-8"))
   ...:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪done extensive testing	749

In [5]: for record in z.search(prefix=b"not done extensive "):
   ...:     print(record.decode("utf-8"))
   ...:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪done extensive research	225
```

```
not done extensive testing        749
not done extensive tests        87

In [6]: for record in z.search(start=b"not done ext", stop=b"not done fast"):
   ...:     print(record.decode("utf-8"))
   ...:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪done extensive research        225
not done extensive testing        749
not done extensive tests        87
not done extremely well        41
not done fairly .        61
```

Error reporting

zs defines two exception types.

exception zs.**ZSError**
Exception class used for most errors encountered in the ZS package. (Though we do sometimes raise exceptions of the standard Python types like `IOError`, `ValueError`, etc.)

exception zs.**ZSCorrupt**
A subclass of *ZSError*, used specifically for errors that indicate a malformed or corrupted ZS file.

Reading

Reading ZS files is accomplished by instantiating an object of type *ZS*:

class zs.**ZS**(*path=None*, *url=None*, *parallelism='guess'*, *index_block_cache=32*)
Object representing a .zs file opened for reading.

Parameters

- **path** – A string containing an on-disk file to be opened. Exactly one of `path` or `url` must be specified.
- **url** – An HTTP (or HTTPS) URL pointing to a .zs file, which will be accessed directly from the server. The server must support Range: queries. Exactly one of `path` or `url` must be specified.
- **parallelism** – When querying a ZS file, there are always at least 2 threads working in parallel: the main thread, where you iterate over the results and presumably do something with them, and a second thread used for IO. In addition, we can spawn any number of worker processes which will be used internally for decompression and other CPU-intensive tasks. `parallelism=1` means to spawn 1 worker process; if you want to perform decompression and other such tasks in serial in your main thread, then use `parallelism=0`. The default of `parallelism="guess"` means to spawn one worker process per available CPU.

  Note that if you know that you are going to read just a few records on each search, then parallelism=0 may be slightly faster; this saves the overhead of setting up the worker processes, and they only really help when doing large bulk reads.
- **index_block_cache** – The number of index blocks to keep cached in memory. This speeds up repeated queries. Larger values provide better caching, but take more memory.

Make sure that this is at least as large as your file's *root_index_level*, or else the cache will be useless.

This object can be used as a context manager, e.g.:

```
with ZS("./my/favorite.zs") as zs_obj:
    ...
```

is equivalent to:

```
zs_obj = ZS("./my/favorite.zs")
try:
    ...
finally:
    zs_obj.close()
```

Basic searches

class `zs.`**ZS**

search(*start=None*, *stop=None*, *prefix=None*)
Iterate over all records matching the given query.

A record is considered to "match" if:

- `start <= record`, and
- `record < stop`, and
- `record.startswith(prefix)`

Any or all of the arguments can be left as `None`, in which case the corresponding check or checks are not performed.

Note the asymmetry between `start` and `stop` – this is analogous to other Python constructs which use half-open [start, stop) ranges, like `range()`.

If no arguments are given, iterates over the entire contents of the .zs file.

Records are always returned in sorted order.

__iter__()
Equivalent to `zs_obj.search()`.

File attributes and metadata

ZS objects provides a number of read-only attributes that give general information about the ZS file:

class `zs.`**ZS**

metadata
A .zs file can contain arbitrary metadata in the form of a JSON-encoded dictionary. This attribute contains this metadata in unpacked form.

root_index_offset
The file offset of the root index block, as stored in the *header*.

root_index_length
The length of the root index block, as stored in the *header*.

total_file_length
The proper length of this file, as stored in the *header*.

codec
The compression codec used on this file, as a byte string.

data_sha256
A strong hash of the underlying data records contained in this file. If two files have the same value here, then they are guaranteed to represent exactly the same data (i.e., return the same records to the same queries), though they might be stored using different compression algorithms, have different metadata, etc.

root_index_level
The level of the root index.

Starting from scratch, finding an arbitrary record in a ZS file requires that we fetch the header, fetch the root block, and then fetch this many blocks to traverse the index tree. So that's a total of `root_index_level + 2` fetches. (On local disk, each "fetch" is a disk seek; over HTTP, each "fetch" is a round-trip to the server.) For later queries on the same *ZS* object, at least the header and root will be cached, and (if you're lucky) other blocks may be as well.

Fast bulk operations

If you want to perform some computation on many records (e.g., all the records in your file), then these functions are the most efficient way to do that.

class `zs.ZS`

block_map (*fn*, *start=None*, *stop=None*, *prefix=None*, *args=()*, *kwargs={}*)
Apply a given function – in parallel – to records matching a given query. This function is lazy – if you don't iterate over the results, then the function might not be called on all of them.

Using this method (or its friend, *block_exec()*) is the best way to perform large bulk operations on ZS files.

The way to think about how it works is, first we find all records matching the given query:

```
matches = zs_obj.search(start=start, stop=stop, prefix=prefix)
```

and then we divide the resulting list of records up into arbitrarily sized chunks, and for each chunk we call the given function, and yield the result:

```
while there are matches:
    chunk = list(get arbitrarily many matches)
    yield fn(chunk, *args, **kwargs)
```

But, there is a trick: in fact many copies of the function are run in parallel in different worker processes, and then the results are passed back to the main process for you to do whatever you want with. (Think "poor-man's map-reduce".)

This means that your `fn`, `args`, `kwargs`, and return values must all be pickleable. In particular, `fn` probably has to either be a global function in a named module, or else an object with a `__call__` method that is an instance of a globally defined class in a named module. (Sorry, I didn't make the rules. Feel free to submit patches to use a more featureful serialization library like 'dill', esp. if you can demonstrate that they don't add too much overhead.)

This will be most efficient if `fn` performs non-trivial work, and especially if it can avoid returning large/complicated structures from `fn` – after all, the whole idea is that the code that's looping over the results from *`block_map()`* should have less work to do than it would if it were just calling *`search()`* directly.

If you manage to take this to the extreme where you have nothing to return from *`block_map()`* (maybe your `fn` is writing to a database or something), then you can use *`block_exec()`* instead to save a bit of boilerplate.

If you pass `parallelism=0` when creating your *`ZS`* object, then this method will perform all work within the main process. This makes debugging a lot easier, because it will let you get real backtraces if (when) your `fn` crashes.

block_exec(*fn*, *start=None*, *stop=None*, *prefix=None*, *args=()*, *kwargs={}*)
Eager version of *`block_map()`*.

This is equivalent to calling *`block_map()`*, iterating over the results, and throwing them all away.

High-level operations

class `zs.`**ZS**

dump(*out_file*, *start=None*, *stop=None*, *prefix=None*, *terminator=b'\n'*, *length_prefixed=None*)
Decompress a given range of the .zs file to another file. This is performed in the most efficient available way.

Parameters

- **terminator** (*byte string*) – A terminator appended to the end of each record. Default is a newline. (Ignored if `length_prefixed` is given.)
- **length_prefixed** – If given, records are output in a length-prefixed format, and `terminator` is ignored. Valid values are the strings `"uleb128"` or `"u64le"`, or None.

See *`search()`* for the definition of `start`, `stop`, and `prefix`.

On Python 3, `out_file` must be opened in binary mode.

For a convenient command-line interface to this method, see *zs dump*.

validate()
Validate this .zs file for correctness.

This method does an exhaustive check of the current file, to validate it for self-consistency and compliance with the ZS specification. It should catch all cases of disk corruption (with high probability), and all cases of incorrectly constructed files.

This reads and decompresses the entire file, so may take some time.

For a convenient command-line interface to this method, see *zs validate*.

Writing

In case you want a little more control over ZS file writing than you can get with the `zs make` command-line utility (see *zs make*), you can also access the underlying ZS-writing code directly from Python by instantiating a *`ZSWriter`* object.

class `zs.`**`ZSWriter`**(*path*, *metadata*, *branching_factor*, *parallelism='guess'*, *codec='lzma'*, *codec_kwargs={}*, *show_spinner=True*, *include_default_metadata=True*)

`add_data_block`(*records*)

Append the given set of records to the ZS file as a single data block.

(See *On-disk layout of ZS files* for details on what a data block is.)

Parameters **`records`** – A list of byte strings giving the contents of each record.

`add_file_contents`(*file_handle*, *approx_block_size*, *terminator=b'\n'*, *length_prefixed=None*)

Split the contents of file_handle into records, and write them to the ZS file.

The arguments determine how the contents of the file are divided into records and blocks.

Parameters

- **`file_handle`** – A file-like object whose contents are read. This file is always closed.
- **`approx_block_size`** – The approximate size of each data block, in bytes, *before* compression is applied.
- **`terminator`** – A byte string containing a terminator appended to the end of each record. Default is a newline.
- **`length_prefixed`** – If given, records are output in a length-prefixed format, and `terminator` is ignored. Valid values are the strings `"uleb128"` or `"u64le"`, or `None`.

`finish`()

Declare this file finished.

This method writes out the root block, updates the header, etc.

Importantly, we do not write out the correct magic number until this method completes, so no ZS reader will be willing to read your file until this is called (see *Magic number*).

Do not call this method unless you are sure you have added the right records. (In particular, you definitely don't want to call this from a `finally` block, or automatically from a `with` block context manager.)

Calls *`close()`*.

`close`()

Close the file and terminate all background processing.

Further operations on this ZSWriter object will raise an error.

If you call this method before calling *`finish()`*, then you will not have a working ZS file.

This object can be used as a context manager in a `with` block, in which case *`close()`* will be called automatically, but *`finish()`* will not be.

`closed`

Boolean attribute indicating whether this ZSWriter is closed.

CHAPTER 4

Conventions

Our experience is that most data sets have various unique features, so ZS is an unopinionated format: we give you a hunk of JSON and a pile of binary records, and let you figure out what to put in them. But, it is nice to have some conventions for how to handle common situations. As more people use the format, these will probably evolve, but for now here are some notes.

Metadata

[XX document the metadata being used in the current gbooks files]:

```
"build-info": {
    "host": "morel.ucsd.edu",
    "user": "njsmith",
    "time": "2014-04-21T23:56:47.225267Z",
    "version": "zs 0.0.0-dev"
},
"corpus": "google-books-eng-us-all-20120701",
"subset": "2gram",
"record-format": {
    "separator": "\t",
    "column-types": [
        "utf8",
        "int",
        "int",
        "int"
    ],
    "type": "separated-values",
    "column-names": [
        "ngram",
        "year",
        "match_count",
        "volume_count"
    ]
}
```

Some other items you might want to consider including:

- Information on the preprocessing pipeline that led to this file (for answering questions like, "is this the version that had case-sensitivity enabled, or disabled?")
- Bibliographic references for papers that users of this data might want to refer to or cite.
- Your contact information.
- Any relevant DOIs or ORCIDs.

Record format

The ZS format itself puts absolutely no limitations on the contents of individual records. You can encode your data any way you feel like. However, because indexing is done by ASCIIbetical sort order, it's probably a good idea to choose an encoding which makes sort order meaningful. Some general principles:

- **Put the field you want to index on first**; if you want to be able to index on multiple fields simultaneously, put them first, second, third, etc.
- **Beware of quoting.** This arises especially for common CSV formats, where fields containing the characters `,` or `"` often get special handling. For example, suppose we have some nicely organized n-grams:

```
to be , or not to be
to be disjoint and out of frame
'Tis Hamlet 's character . " Naked ! "
'Tis now the very witching time
What a piece of work is a man !
```

  If we encode these as a column in CSV format, and then sort, we end up with:

```
"'Tis Hamlet 's character . "" Naked ! """
"to be , or not to be"
'Tis now the very witching time
What a piece of work is a man !
to be disjoint and out of frame
```

  Notice that every entry that contained a `,` or `"` has been wrapped in `"`'s. If we want to find n-grams beginning `to be` or `'Tis` then a simple prefix search will no longer work; when we want to find records with the prefix `foo` we have to remember always to search for both `foo` and `"foo`.

  Ideally there is some character that you know will never occur in any field, and you can use that for your separator – then no quoting is ever needed. This might be tab (t), or if you get desperate then there are other options like NUL (00) or newline (n) – though with these latter options you'll lose some of the convenience of browsing your data with simple tools like *zs dump*, and may have to play around a bit more with *zs make*'s options to construct your file in the first place.

  Alternatively, other quoting schemes (e.g., replacing `,` with `\\,` and `\\` with `\\\\`) may not perfectly preserve sorting, but they do preserve prefix searches, which is often the important thing.

- **Beware of standard number formats.** String-wise, `"10"` is less than `"2"`, which is a problem if you want to be able to do range queries on numeric data in ZS files. Some options for working around this include using fixed-width strings (`"10"` and `"02"`), or using some kind of big-endian binary encoding. Note that the ASCII space character (0x20) sorts before all printing characters, including digits. This means that instead of padding with zeroes like in `"02"`, it will also work to pad with spaces, `" 2"`. Fixed width formats in general can be cumbersome to work with, but they do have excellent sorting properties.

In the Google n-grams, the year field fortunately turns out to be fixed width (at least until Google starts scanning papyruses). And for the actual count fields, this formatting issue doesn't arise, because we have no reason to index on them.

- **Beware of little-endian Unicode and surrogate pairs.** ASCII, UTF-8, and UTF-32BE all have sensible sort orders (i.e., ASCIIbetical sort on the encoded strings is the same as lexicographic sort on code points). This is definitely not true for UTF-16LE or UTF-32LE, and is not *quite* true for UTF-16BE, because of the existence of surrogate pairs (see e.g.).

  Of course, if all you want are exact prefix searches, then these issues don't really matter.

  We recommend using UTF-8 unless you have a good reason not to.

  Note that the `zs` command-line tool has a mild bias towards UTF-8, in that if you pass it raw Unicode characters for `--start`, `--stop`, or `--prefix`, then it encodes them as UTF-8 before doing the search.

If these issues turn out to cause enough problems, it may makes sense at some point to define a revised version of the ZS format which has an explicit schema for record contents, and uses a content-sensitive sort order (e.g., one that performs numeric comparison on numeric fields).

CHAPTER 5

Publically available ZS datasets

[XX expand]

The v2 Google Books eng-us-all n-grams are here, at least temporarily: http://bolete.ucsd.edu/njsmith/

Note that these files use "0gram" to refer to what Google calls "totalcounts", thus preserving the rule that n-gram counts are normalized by (n-1)-gram counts. The simple dependency arcs that Google calls "0grams" are not included, since they seem to have been superseded by later data releases.

The zscontrib repository has some scripts to fetch an arbitrary v2 Google Books sub-corpus and build it into a set of .zs files.

CHAPTER 6

On-disk layout of ZS files

This page provides a complete specification of version **0.10** of the ZS file format, along with rationale for specific design choices. It should be read by anyone who plans to implement a new reader or writer for the format, or is just interested in how things work under the covers.

Overview

ZS is a read-only database format designed to store a multiset of records, where each record is an uninterpreted string of binary data. The main design goals are:

- Locating an arbitrary record, or sorted span of records, should be fast.
- Doing a streaming read of a large span of records should be fast.
- Hardware is unreliable, especially on the scale of terabytes and years, and ZS is designed for long-term archival of multi-terabyte data. Therefore it must be possible to quickly and reliably validate the integrity of the data returned by every operation.
- It should be reasonably efficient to access files over slow, "dumb" transports like HTTP.
- Files should be as small as possible while achieving the above goals.

The main complication influencing ZS's design is that compression is necessary to achieve reasonable storage sizes, but decompression is slow, block-oriented, and inherently serial, which puts the last goal in direct conflict with the first two. Compressing a chunk of data is like wrapping it up into an opaque bundle. The only way to find something inside is to first unwrap (decompress) the whole thing. This is why it won't work to simply write our data into a large text file and then use a standard compression program like `gzip` on the whole thing. If we did this, then the only way to find any piece of data would be to decompress the whole file, which takes ages. Instead, we need some way to split our data up into multiple smaller bundles. Once we've done this, reading individual records can be fast, because we only have to unwrap a single small bundle, not a huge one. And, it turns out, splitting up our data into multiple bundles also makes bulk reads faster. For a large read, we have to unpack the same amount of total data regardless of whether it's divided into small bundles or not, so the total work is constant. But, in the multiple-bundle case, we can easily divvy up this work across multiple CPUs, and thus finish the job more quickly. So, small bundles are great – but, they also have a downside: if we make our bundles too small, then the compression algorithm won't be able to find many

redundancies to compress out, and so our compression ratio will not be very good. In particular, trying to compress individual records would be hopeless.

Our solution is to bundle records together into moderately-sized blocks, and then compress each block. Then we add some framing to let us figure out where each block starts and ends, and add an index structure to let us quickly find which blocks contain records that match some query, and ta-da, we have a ZS file. The resulting structure looks like this:

Fast lookup for arbitrary records is supported by a tree-based indexing scheme: the header contains a pointer to the "root" index block, which in turn refers to other index blocks, which refer to other index blocks, until eventually the lowest-level index blocks refer to data blocks. By following these links, we can locate any arbitrary record in $O(\log n)$ time.

In addition, we require data blocks to be arranged in sorted order within the file. This allows us to do streaming reads starting from any point, which makes for nicely efficient disk access patterns. And range queries are supported by combining these two access strategies: first we traverse the index to figure out which blocks contain records that fall into our range, and then we do a streaming read across these blocks.

General notes

Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Checksumming

To achieve our data integrity goals, every byte in a ZS file that could possibly contain undetected corruption is protected by a 64-bit CRC. Specifically, we use the same CRC-64 calculation that the .xz file format does. The Rocksoft model parameters for this CRC are: polynomial = 0x42f0e1eba9ea3693, reflect in = True, init = 0xffffffffffffffff, reflect out = True, xor out = 0xffffffffffffffff, check = 0x995dc9bbdf1939fa.

Integer representations

Within the ZS header, we make life easier for simple tools like file by encoding all integers using fixed-length 64-bit little-endian format (`u64le` for short).

Outside of the header, integers are encoded in the *uleb128* format, familiar from the DWARF debugging format. Okay, maybe not so familiar. This is a simple variable-length encoding for unsigned integers of arbitrary size using **u**nsigned **l**ittle-**e**ndian **b**ase-**128**. To read a uleb128 value, you proceed from the beginning of the string, one byte at a time. The lower 7 bits of each byte give you the next 7 bits of your integer. This is little-endian, so the first byte gives you the least-significant 7 bits of your integer, then the next byte gives you bits 8 through 15, the one after that the bits 16 through 23, etc. The 8th, most-significant bit of each byte serves as a continuation byte. If this is 1, then you keep going and read the next byte. If it is 0, then you are done. Examples:

```
uleb128 string  <->  integer value
--------------       -------------
            00                0x00
            7f                0x7f
         80 01                0x80
```

```
         ff 20              0x107f
80 80 80 80 20             2 ** 33
```

(This format is also used by protocol buffers, by the XZ file format, and others.) This format allows for redundant representations by adding leading zeros, e.g. the value 0 could also be written `80 00`. However, doing so is forbidden; all values MUST be encoded in their shortest form.

Layout details

Here's the big picture – refer to it while reading the full details below.

ZS files consist of a *magic number*, followed by a *header*, followed by a sequence of *blocks*. Blocks come in two types: *data blocks*, and *index blocks*.

Magic number

To make it easy to distinguish ZS files from non-ZS files, every valid ZS file begins with 8 magic bytes. Specifically, these ones (written in hex, with ASCII below):

```
ab 5a 53 66 69 4c 65 01   # Good magic
    Z  S  f  i  L  e
```

If there's ever an incompatible ZS version 2, we'll use the last byte as a version number.

Writing out a large ZS file is an involved operation that might take a long time. It's possible for a hardware or software problem to occur and cause this process to be aborted before the file is completely written, leaving behind a partial, corrupt ZS file. Because ZS is designed as a reliable archival format we would like to avoid the possibility of confusing a corrupt file with a correct one, and because writing ZS files can be slow, after a crash we would like to be able to reliably determine whether the writing operation completed, and whether we can trust the file left behind. Therefore we also define a second magic number to be used specifically for partial ZS files:

```
ab 5a 53 74 6f 42 65 01   # Bad magic
    Z  S  t  o  B  e
```

It is RECOMMENDED that ZS file writers perform the following sequence:

- Write out the `ZStoBe` magic number.
- Write out the rest of the ZS file.
- Update the header to its final form (including, e.g., the offset of the root block).
- (IMPORTANT) Sync the file to disk using `fsync()` or equivalent.
- Replace the `ZStoBe` magic number with the correct `ZSfiLe` magic number.

Following this procedure guarantees that, modulo disk corruption, any file which begins with the correct ZS magic will in fact be a complete, valid ZS file.

Any file which does not begin with the correct ZS magic is not a valid ZS file, and MUST be rejected by ZS file readers. Files with the `ZStoBe` magic are not valid ZS files. However, polite ZS readers SHOULD generally check for the `ZStoBe` magic, and if encountered, provide an informative error message while rejecting the file.

Header

The header contains the following fields, in order:

- Length (`u64le`): The length of the data in the header. This does not include either the length field itself, or the trailing CRC – see diagram.
- Root index offset (`u64le`): The position in the file where the root index block begins.
- Root index length (`u64le`): The number of bytes in the root index block. This *includes* the root index block's length and CRC fields; the idea is that doing a single read of this length, at the given offset, will give us the root index itself. This is an important optimization when IO has high-latency, as when accessing a ZS file over HTTP.
- Total file length (`u64le`): The total number of bytes contained in this ZS file; the same thing you'd get from `ls -l` or similar.

  > **Warning:** To guarantee data integrity, readers MUST validate the file length field; our CRC checks alone cannot detect file truncation if it happens to coincide with a block boundary.

- SHA-256 of data (32 bytes): The SHA-256 hash of the stream one would get by extracting all data block payloads and concatenating them. The idea is that this value uniquely identifies the logical contents of a ZS file, regardless of storage details like compression mode, block size, index fanout, etc.
- Codec (16 bytes): A null-padded ASCII string specifying the codec (compression method) used. Currently defined codecs include:
  - `none`: Block payloads are stored in raw, uncompressed form.
  - `deflate`: Block payloads are stored using the deflate format as defined in RFC 1951. Note that this is different from both the gzip format (RFC 1952) and the zlib format (RFC 1950), which use different framing and checksums. ZS provides its own framing and checksum, so we just use raw deflate streams.
  - `lzma2;dsize=2^20`: Block payloads are represented as raw LZMA2 bitstreams that can be decompressed using a dictionary size of 2^20 bytes (i.e., 1 MiB); this means that each decoder needs an upper bound of ~2 MiB of memory. Note that while it might look parametrized, this is a simple literal string – for example, using the encoder string `lzma2;dsize=2^21` is illegal. This means you can use the standard XZ presets 0 and 1, including the "extreme" 0e and 1e modes, but not higher. This is pretty reasonable, since there is never any advantage to using a dictionary size that is larger than a single block payload, and we expect >1 MiB blocks to be rare; but, if there is demand, we may add further modes with larger dictionary sizes.

    As compared to using XZ format, raw LZMA2 streams are ~0.5% smaller, so that's nice. And, more importantly, the use of raw streams dramatically reduces the complexity requirements on readers, which is important for an archival format. Doing things this way means that readers don't need to be prepared to handle the multi-gigabyte dictionary sizes, complicated filter chains, multiple checksums, etc., which the XZ format allows.
- Metadata length (`u64le`): The length of the next field:
- Metadata (UTF-8 encoded JSON): This field allows arbitrary metadata to be attached to a ZS file. The only restriction is that the encoded value MUST be what JSON calls an "object" (also known as a dict, hash table, etc. – basically, the outermost characters have to be `{}`). But this object can contain arbitrarily complex values (though we recommend restricting yourself to strings for the keys). See *Metadata*.
- <extensions> (??): Compliant readers MUST ignore any data occurring between the end of the metadata field and the end of the header (as defined by the header length field). This space may be used in the future to

add backwards-compatible extensions to the ZS format. (Backwards-incompatible extensions, of course, will include a change to the magic number.)

- CRC-64-xz (`u64le`): A checksum of all the header data. This does not include the length field, but does include everything between it and the CRC. See diagram.

Blocks

Blocks themselves all have the same format:

- Length (`uleb128`): The length of the data in the block. This does not include either the length field itself, or the trailing CRC – see diagram.
- Level (`u8`): A single byte encoding the "level" of this block. Data blocks are level 0. Index blocks can have any level between 1 and 63 (inclusive). Other levels are reserved for future backwards-compatible extensions; compliant readers MUST silently ignore any block with its level field set to 64 or higher.
- Compressed payload (arbitrary data): The rest of the block after the level is a compressed representation of the payload. This should be decompressed according to the value of the codec field in the header, and then interpreted according to the rules below.
- CRC-64-xz (`u64le`): CRC of the data in the block. This does not include the length field – see diagram. Note that this is calculated directly on the raw disk representation of the block, compression and all.

Technically we don't need to store the length at the beginning of each block, because every block also has its length stored either in an index block or (for the root block) in the header. But, storing the length directly at the beginning of each block makes it much simpler to write naive streaming decoders, reduces seeks during streaming reads, and adds negligible space overhead.

Data block payload

Data block payloads encode a list of records. Each record has the form:

- Record length (`uleb128`): The number of bytes in this record.
- Record contents (arbitrary data): That many bytes of data, making up the contents of this record.

Then this is repeated as many times as you want.

Every data block payload MUST contain at least one record.

Index block payload

Index block payloads encode a list of references to other index or data blocks.

Each index payload entry has the form:

- Key length (`uleb128`): The number of bytes in the "key".
- Key value (arbitrary data): That many bytes of data, making up the "key" for the pointed-to block. (See below for the invariants this key must satisfy.)
- Block offset (`uleb128`): The file offset at which the pointed-to block is located.
- Block length (`uleb128`): The length of the pointed-to block. This *includes* the root index block's length and CRC fields; the idea is that doing a single read of this length, a the given offset, will give us the root index itself. This is an important optimization when IO has high-latency, as when accessing a ZS file over HTTP.

Then this is repeated as many times as you want.

Every index block payload MUST contain at least one entry.

Key invariants

All comparisons here use ASCIIbetical order, i.e., lexicographic comparisons on raw byte values, as returned by `memcmp()`.

We require:

- The records in each data block payload MUST be listed in sorted order.
- If data block A occurs earlier in the file (at a lower offset) than data block B, then all records in A are REQUIRED to be less-than-or-equal-to all records in B.
- Every block, except for the root block, MUST be referenced by exactly one index block.
- An index block of level n MUST only reference blocks of level $n - 1$. (Data blocks are considered to have level 0.)
- The keys in each index block payload MUST occur in sorted order.
- To every block, we assign a span of records as follows: data blocks span the records they contain. Index blocks span all the records that are spanned by the blocks that they point to (recursively). Given this definition, we can state the key invariant for index blocks: every index key MUST be less-than-or-equal-to the *first* record which is spanned by the pointed-to block, and MUST be greater-than-or-equal-to all records which come before this record.

  Note: According to this definition, it is always legal to simply take the first record spanned by a block, and use that for its key. But we do not guarantee this; advanced implementations might take advantage of this flexibility to choose shorter keys that are just long enough to satisfy the invariant above. (In particular, there's nothing in ZS stopping you from having large individual records, up into the megabyte range and beyond, and in this case you might well prefer not to copy the whole record into the index block.)

Notice that all invariants use non-strict inequalities; this is because the same record might occur multiple times in different blocks, making strict inequalities impossible to guarantee.

Notice also that there is no requirement about where index blocks occur in the file, though in general each index will occur after the blocks it points to, because unless you are very clever you can't write an index block until after you have written the pointed-to blocks and recorded their disk offsets.

Specification history

- Version 0.10: Remove support for the `bz2` compression format.
- Version 0.9: First public release.

CHAPTER 7

Release history

v0.10.0

Switched the default codec to LZMA, and dropped bzip2 support, since LZMA seems as good or better on pretty much all axes.

Upped the default `--approx-block-size` to 384 KiB.

Improved `zs make`'s spinner output.

v0.9.0

First public release.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

z

Index

Symbols

A

B

C

D

F

M

R

S

T

V

Z