ZS: A file format for efficiently distributing, using, and archiving record-oriented datasets of any size

Nathaniel J. Smith

School of Informatics, University of Edinburgh

DRAFT of August 24, 2014

Author Note

This work was supported in part by NIH grant T32-DC000041 to the Center for Research in Language, NIH grant T32-MH20002 to the Institute for Neural Computation, funding from the Army Research Laboratories Cognition & Neuroergonomics Collaborative Technology Alliance, and funding from the European Union's 7th Framework Programme under grant agreement No. 270273 (XPERIENCE). Thanks to: R. Levy for comments and discussion, and E. Van Everbroeck for superior sysadminship and timely cluster reboots.

Abstract

Behavioral research increasingly benefits from the use of very large datasets like the n-gram corpora released by Google. While such datasets provide unprecedented scientific opportunities, their sheer scale also creates unprecedented technical challenges which make them inaccessible to many potential users. Here we introduce the ZS file format, a method for storing arbitrary record-oriented data in compressed form. The ZS format is easy to implement and even easier to use, but is highly effective: when converting the 2012 Google Books n-grams (Y. Lin et al., 2012) from Google's current distribution format to ZS, the files become 41% smaller, yet many important query operations now take milliseconds instead of hours or days, thus enabling the kind of interactive data exploration that is often key to building scientific insight. ZS files also support advanced features like multicore decompression, embedded metadata, automatic end-to-end data integrity checking, and the ability to perform queries directly over the web without downloading a full dataset and without any special software on the server, which allows technically unsophisticated users with ordinary personal computers to take advantage of even the largest datasets. Of course, small datasets also benefit from such features; ZS files make a superior alternative to tab- or comma-separated value files for distributing, using, and archiving datasets of any size. We release both free software for working with ZS files and a version of the 2012 Google n-grams in ZS format.

ZS: A file format for efficiently distributing, using, and archiving record-oriented datasets of any size

Introduction: Big data is a problem

Eight years ago, Brants and Franz (2006) released the English Web 1T corpus, which provided counts of how often various n-grams (one to five word phrases) occurred in one trillion words of web text. This dataset created unique new opportunities – and unique new problems. On the one hand, it gave ordinary researchers access to statistics about language use on an unprecedented scale: the Web 1T is based on 1,000,000 times more text than the well-known Kučera and Francis (1967) norms, and 10,000 times more text than the British National Corpus. This scale enabled a wide variety of novel behavioural and neuropsychological investigations into the interaction of linguistic statistics and cognitive processing (e.g., Mitchell et al., 2008; Piantadosi, Tily, & Gibson, 2011; Shaoul, Westbury, & Baayen, 2013; Smith & Levy, 2011). But, on the other hand, the size of the corpus causes severe technical challenges. The raw data is represented as a collection of text files, where each line of text contains a single n-gram and its count. Together, these text files amount to 93 GB[1] of data. Merely searching 93 GB of data is a non-trivial operation, but to make matters worse, these files are not distributed directly; instead, they are first subjected to gzip compression. This has the advantage of reducing the size to a 'mere' 26 GB, which is a substantial boon for distribution – it allows the data to fit onto 6 DVDs instead of 20. But it also has the disadvantage of making even simple tasks prohibitively cumbersome; looking up the count for any single n-gram now requires us to first decompress tens of megabytes of data. Researchers who wish to make use of this dataset in any systematic way must therefore first identify and implement some strategy to store and search it.

The choice of such a strategy depends on both the researcher's goals and their available resources. We start by considering the simplest approach, which is this: whenever

[1]We will use GB and TB to refer to base-10 gigabytes and terabytes, so that 1 TB = 1000 GB, not base-2 gibibytes (GiB) and tebibytes (TiB).

we want to find some the records matching some pattern, we perform a linear read through all the data (possibly decompressing on the fly), and make a note of the records which match. Hawker, Gardiner, and Bennetts (2007) and Bansal and Klein (2011) describe tools for performing this type of query on the Web 1T corpus, with the additional optimization that they allow many queries to be specified up front, and then answered together by performing a single linear scan of the data. They report processing 10^6 queries against the 5-gram subset of the Web 1T in 1 hour, and 4.5×10^6 queries against the full Web 1T in 115 minutes, respectively. This batching approach can handle arbitrarily complicated queries, requires only as much disk space as is needed to store the compressed data, and can achieve a high rate of query throughput on average. But the important words here are "on average"; achieving high throughput relies on amortizing the query overhead across a large number of queries. If we only have one query to run, it will still take an hour or more to get our results. Such slow turnaround is a major obstacle to the interactive data exploration that is often key to building scientific insight. Can we do better?

A simple and popular way to speed things up is to store our data on disk as sorted, decompressed text files (possibly after some preprocessing) and then use binary search to efficiently find particular lines of interest. This works well in cases where we can afford to store the decompressed data, and where we only need to find specific n-grams – e.g., the 3-gram `this is fun` – or n-grams based on their initial prefix – e.g., all 3-grams matching `this is *` – which is already enough to support many psycholinguistically important applications, like computation of transitional probabilities (McDonald & Shillcock, 2003a, 2003b), n-gram frequencies (Arnon & Snider, 2010; Bannard & Matthews, 2008; Shaoul et al., 2013; Tremblay, Derwing, Libben, & Westbury, 2011), and n-gram based surprisal estimates (Smith & Levy, 2011, 2013). If we wish to support more complex queries like finding collocations (all words that co-occur with `fun`) or matching arbitrary templates (like `this * fun`, or `* is *`), then more sophisticated strategies are needed. Carlson and Fette (2007) and Evert (2010) describe approaches that allow such queries to be performed interactively after first loading the Web 1T data into an off-the-shelf relational database system. These databases were able to answer complex queries in seconds or minutes instead of hours, but the trade-off is that they required well over 200 GB of storage space (i.e., more than doubling the size of the already uncompressed text files), and Evert reports that his database took two weeks to construct.

These limitations of generic text files and databases have motivated several groups to develop sophisticated tools specifically for storing and working with n-gram data, some of which have focused on answering various types of linguistically rich queries in reasonable time (Flor, 2013; D. Lin et al., 2010; Sekine, 2008; Sekine & Dalwani, 2010), while others have spent intense engineering effort on producing compact data structures for use in natural language processing applications that allow individual records to be found as quickly as possible (Germann, Joanis, & Larkin, 2009; Heafield, 2011; Kaldager, 2012; Pauls & Klein, 2011).

Eight years later, in 2014, the English Web 1T dataset is far from unique; big data, and the associated challenges, are increasingly the order of the day. Similar web-based n-gram counts have been released for a variety of languages (Brants & Franz, 2009; Liu, Yang, & Lin, 2010; Kudo & Kazawa, 2009), and the Google Books group has released multiple datasets based on scanned books (which produce far cleaner data than web text): first, in 2010, a set of multilingual counts subdivided by year of publication and at several levels of granularity (e.g., providing both the total number of times each n-gram was observed, and the number of different books it was observed in; Michel et al., 2011[2]). This release was later supplemented by detailed syntactic dependency statistics for English (Goldberg & Orwant, 2013). A second release followed in 2012 with a larger corpus, and the addition of part-of-speech tags and basic syntactic statistics for all languages (Y. Lin et al., 2012). These additional layers of data beyond raw counts open up new possibilities for

[2]Confusingly, this release represents a snapshot of the Google Books scanning project that was taken 2009, released publically in 2010, and described in a 2011 publication. In this historical summary we'll refer to this as the 2010 release, since this is when it became generally available.

behavioral research, such as tracking diachronic changes in syntactic usage or estimating per-participant frequency counts based on books released during each individual's lifespan. But they also cause a massive increase in the size of the data. Fig. 1 shows the raw (uncompressed) and as-distributed (compressed) sizes for three English n-gram datasets; the leftmost bars represent the Web 1T English corpus whose size caused so much trouble above.

How can we work with a massive dataset like the 2012 Google Books n-grams? None of the strategies described above are very helpful. Storing the files compressed with a standard tool like gzip is easy to do (since that's how the files are distributed in the first place), but then the only way to access the data is via linear scans. If we have many queries to perform then it is possible to use the batching strategies cited above to amortize the cost of a linear scan, but a crude extrapolation of their quoted results from the Web 1T to the 2012 Google Books corpus suggests that such a batch query will still take days or weeks to run. (Simply decompressing the data set as currently distributed, a necessary first step in any processing, already requires more than thirty hours of CPU time.[3]) Alternatively, we could store the uncompressed text, but this requires 20+ TB of storage, which is prohibitive for most researchers – never mind doubling the size by loading it into a relational database. Or, we could convert our files to some other format that takes advantage of specialized knowledge of this data to efficiently pack the individual records into minimal space while still allowing for fast queries – but it turns out that none of the specialized tools and formats described above can be used directly, because they are designed to store n-gram + count data, while the Google Books files contain n-gram + year + multiple counts. Even if the engineering effort were invested to design new specialized tools and file formats, then their usage would still be restricted to the minority of researchers who have both the technical sophistication to select a format appropriate to

[3]All performance metrics quoted in this paper are intended only to illustrate general trends and relative orders of magnitude, so we do not report detailed methodology; for reference, measurements were made on a 16-core compute server purchased in 2012.

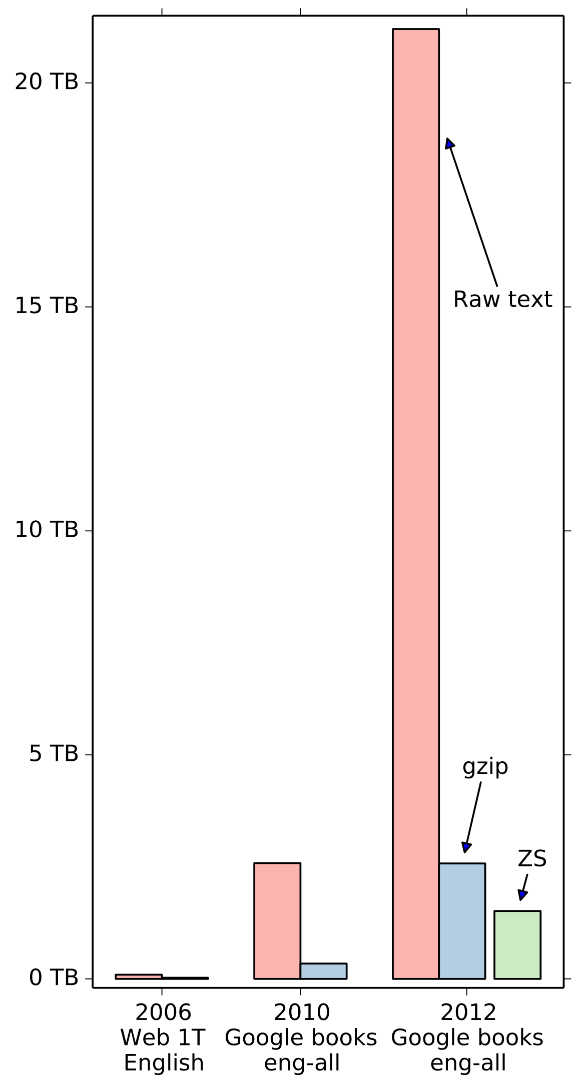

Figure 1. English n-grams: six years of progress in preposterously large datasets. Shown are the raw (uncompressed) and as-distributed (compressed) sizes for three datasets. Each of these datasets became, at its time of release, the largest and most detailed publically available source of information on word co-occurrence statistics in English. For comparison, the 2012 Google Books `eng-all` dataset is also shown stored using the ZS format with default settings.

their particular needs, and the computing power needed to decompress the data and repack it into this format. And then we'd have to repeat the whole exercise every time a new dataset is released.

Here, we take a different approach. Rather than proposing a new tool and/or format that is specialized for *using* the n-gram data in some particular way, we focus on designing the best possible *distribution* format for a broad range of datasets: the ZS format. The motivation is that if we build a better distribution format, we ensure that the maximum number of users will benefit. Everyone uses distribution formats – even if only as input to a batch query system or as a starting point for conversion to a format that is specialized to their particular needs – and since format conversions for large datasets can be prohibitively costly, for many users the distribution format may be the only format which is practicably accessible. As compared to the current de facto standard – gzipped text files – ZS files are generally smaller, faster, or both. In addition, they provide better data integrity guarantees, structured metadata, and a limited indexing capability allowing for prefix searches. ZS files even allow index lookups and partial downloads to be performed directly against any file which is available for download over the web, without the need for any special software to be installed on the server; this makes it possible to download only the relevant portion of a large remote dataset, or to query large datasets from computers whose storage is too limited to hold even the compressed files. Together, these features mean that for many purposes, the ZS format is not just the best available distribution format, but also a superior format for day-to-day usage and long-term archival.

As discussed above, there are important cases for which prefix searches are inadequate (e.g. locating n-grams matching `this * fun` or `* is *`). But the indices needed to perform such queries would be very large, irrelevant to many users, and require special knowledge of n-grams to be included in the file format; these considerations make them inappropriate in a general purpose distribution format. By contrast, the ZS format's indexing takes minimal space, yet enables many real-world applications (e.g., locating specific records, or finding all continuations of a given n-gram) and is directly applicable to a wide range of data sets.

While its development was motivated by the n-gram datasets' exuberant growth and resulting problems, the ZS format is generic: the only requirement it imposes on a dataset is that it can be represented as a set of *records*, which are treated as opaque binary strings and which correspond to the individual lines of traditional text formats like tab- or comma-separated value files. Thus any TSV or CSV file can be directly converted to or from the ZS format.[4] This is slightly less flexible than gzip, which can be applied to any file whatsoever, but it is key to ZS file's ability to efficiently access individual records. And the restriction does not seem too onerous: all of the n-gram datasets described above meet this requirement, as do many other popular datasets such as the MRC psycholinguistic database (Wilson, 1988), CELEX (Baayen, Piepenbrock, & Gulikers, 1995), the Subtlex word frequency norms (Brysbaert & New, 2009), the data from megastudies like the English Lexicon Project (Balota et al., 2007), etc., meaning that if desired, these datasets could also take advantage of the ZS format's small size, fast lookups, and advanced data integrity guarantees.

Example usage

The reference implementation of the ZS format is distributed as command-line tools and a Python library, available from `http://vorpus.org/ZS`. This package ships with a detailed manual, which we won't try to reproduce here, but to give the flavor of working with ZS files, here's a brief example. Suppose we have a sorted text file `4grams.txt`, containing lines like:

```
this is fun and     2008    5       5
this is fun and     2009    14      14
this is fun for     1858    1       1
this is fun for     1889    2       2
```

[4]Though, as discussed below, some care should be taken with CSV files that contain quoted fields.

The first line, for example, indicates that in Google's database of books published in 2008, the phrase `this is fun and` occurred a total of 5 times across 5 different books. We can convert this text file to a ZS file by running `zs make '{"corpus": "example"}' 4grams.txt 4grams.zs`. The quoted argument at the beginning allows us to specify arbitrary metadata that will be stored into the file, and is discussed in more detail below; the other two arguments simply name the input and output files. To recover our original text, we run: `zs dump 4grams.zs`. The `zs dump` command also has the ability to efficiently extract a specific subset of our file: `zs dump --prefix="this is fun and\t" 4grams.zs` will efficiently retrieve just the lines which begin with the given four words followed by a tab character. Additionally, most of the `zs` commands can be used either with regular local files or with files posted on a web server. For example, if we want to know how often `this is fun` was used in Google-scanned English books in 1955, we don't have to download the full dataset; we can just run:

```
$ zs dump --prefix="this is fun\t1955\t" \
    http://cpl-data.ucsd.edu/zs/google-books-20120701/eng-us-all/google-books-eng-us-all
this is fun     1955    22      22
```

If we want to see all years, we can run:

```
$ zs dump --prefix="this is fun\t" \
    http://cpl-data.ucsd.edu/zs/google-books-20120701/eng-us-all/google-books-eng-us-all
this is fun     1827    2       1
this is fun     1841    3       3
 [... 132 lines suppressed ...]
this is fun     2007    339     324
this is fun     2008    413     402
```

And if we want to see all recorded continuations of `this is`, we can run:

```
$ zs dump --prefix="this is " \
    http://cpl-data.ucsd.edu/zs/google-books-20120701/eng-us-all/google-books-eng-us-all
this is !       1698    2       2
this is !       1714    2       2
 [... 50 MB of data suppressed ...]
```

These commands all complete in seconds, download only the minimum required data from the server, and require no special software on the server side.

There are also options that allow `zs make` and `zs dump` to consume and output files in more exotic (non-text-based) formats; the ZS format itself allows for records to contain arbitrary binary values, including special characters like newlines or null bytes. Other commands include `zs info`, which displays a variety of metadata about the file (including user-specified metadata), and `zs validate`, which can be used to test a file to ensure both that it is well-formed, and that no data has been corrupted by hardware or software errors.

The ZS format

Overview

Because they are intended to be used for interchange and archival, ZS files have a rather simple structure. A complete description is given in the Appendix, but here we give a brief overview of the design, to aid readers in understanding the format's strengths and weaknesses.

A ZS file consists of three types of entities, which are concatenated: the *magic number*, followed by the *header*, followed by a sequence of *blocks*, which are further categorized into *data blocks* and *index blocks* (see Fig. 2).

The magic number is simply a fixed string of eight bytes, which is used to identify whether a given file is in fact in ZS format, and if it is then whether it has been completely written. (Since compressing and writing out large datasets can take hours or days, and

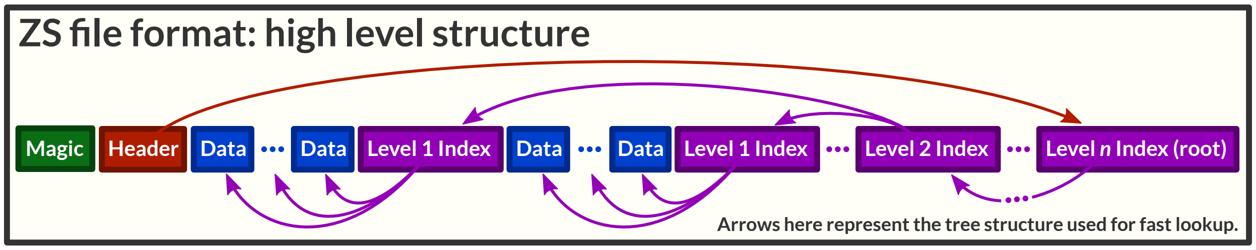
ZS file format: high level structure
Magic
Header
Data
•••
Data
Level 1 Index
Data
•••
Data
Level 1 Index
•••
Level 2 Index
•••
Level n Index (root)
Arrows here represent the tree structure used for fast lookup.

Figure 2. The overall layout of a ZS file. A magic number (used to make it easy to recognize ZS files) is followed by a header containing metadata, and then the sorted data records, which are divided into blocks, with each block individually compressed and checksummed. Index blocks define a lookup tree over the data blocks, so that by walking this tree we can locate an arbitrary span of data with a minimum of disk seeks.

computers do occasionally crash, it's useful to be able to robustly determine whether a file finished writing or not.)

The header contains bookkeeping information needed to access the rest of the file, as well as useful metadata that we will discuss in more detail below.

The data itself is first arranged into a sequence of records, in lexicographically sorted order, and this sequence is then split up into contiguous groups. Each group is packed into a separate data block, and each block is compressed and checksummed individually. This grouping is one of the key ideas underlying the ZS format, and is motivated by fundamental facts about data compression. The goal of a compression algorithm is to discard redundant information, which requires some model of which information is redundant and which is not. There are two ways an algorithm can acquire such a model: it can be hard-coded, as in n-gram storage formats that exploit specialized knowledge of the structure of n-grams to achieve minimal file size (Flor, 2013; Germann et al., 2009; Heafield, 2011; Pauls & Klein, 2011). Or, it can be derived on the fly by examining the data itself, as in the general-purpose compression algorithms used by tools like gzip or xz (Collin, n.d.). Since our goal is to produce a general-purpose file format, we concentrate on the latter approach. General-purpose algorithms can achieve impressive compression ratios – e.g., simply compressing the Web 1T data with xz produces smaller files than four out of the five increasingly complicated special-purpose storage formats proposed by Pauls and

Klein (2011). But, this performance depends on being able to 'see' a substantial quantity of data at once – a special-purpose format can hard-code which aspects of the data are redundant, but a general-purpose algorithm must effectively re-discover these regularities from scratch every time it is run. This creates a fundamental trade-off: we could use a general-purpose algorithm to compress each record separately, which would make it easy to quickly find and decompress any particular record – but our compression ratio would be terrible, because the algorithm could not discover and exploit similarities between adjacent records. On the other hand, if we naively compressed the whole file at once – as in current distribution formats – then the only way to find any particular record would be to decompress the whole file, which can take many minutes of CPU time. Dividing our data into individually-compressed blocks lets us achieve a happy medium: we can make our blocks large enough to achieve effective compression with general-purpose algorithms, but small enough that when we want to locate an individual record, we can decompress the block containing it in milliseconds instead of minutes.

Locating an individual record requires locating the data block which contains it. This is made possible by the index blocks, each of which contains a list of pointers to either other index blocks, or directly to data blocks, along with information about what records can be found by following each pointer. Together the index blocks form a tree, and a pointer to the root of this tree is stored in the header. Traversing this tree allows us to efficiently find all records that fall within a given lexicographic span. This is the main indexing operation supported by ZS files, and in particular includes prefix searches as a special case. This combination of sorted storage and tree-based indexing is a well-known data structure for storing sets; the name "ZS" is an abbreviation of "compressed set".

In the current release of the ZS reference implementation, each index block by default contains 1024 pointers, which means that if we have n data blocks then we can locate an arbitrary one in $\log_{1024} n$ disk seeks. This is ~10x faster than binary search, which requires $\log_2 n$ disk seeks. These index blocks increase the size of the file by a negligible amount, typically less than 0.1% in practice. Each data block on average adds slightly over one index entry, and our need to use relatively large data blocks for compression has the side-effect of guaranteeing that this index entry will be dwarfed by the data block that it points to.

The ZS format also contains provisions for adding new header fields and block types in a backwards compatible manner, should this prove necessary.

Features

Table 1 summarizes the relative performance of three different methods of storing and accessing the English (`eng-all`) portion of the 2012 Google Books n-gram release. We discuss each entry in more detail below.

File size. ZS's storage strategy is designed to make no assumptions about the underlying compression algorithm, which makes it easy to support multiple options. Currently, we support three: deflate (the algorithm used by gzip), LZMA (the algorithm used by xz), and no compression at all. LZMA is the default; this modern algorithm manages to achieve ∼40% better compression than gzip on the 2012 Google Books.

Bulk read speed. Bulk read speed is important: no matter how clever the distribution format, there will be users who need to simply decompress the full dataset and process it into some other form, and the rate at which we can extract decompressed data from our storage format places an upper bound on the speed of such operations.

The cost of LZMA's improved compression ratio is relatively higher CPU usage. For bulk reads, the "As distributed" and "ZS" columns are both CPU-bound, and we see that with a single CPU, gzip on this data decompresses about 4x faster than LZMA. But this is not the whole story: standard decompression algorithms are intrinsically serial, and thus unable to exploit multiple CPUs; standard tools like gzip and xz then inherit this limitation. But because ZS divides data up into many independently-compressed blocks, each block can trivially be decompressed in parallel. Providing 8 CPUs doesn't help the

Table 1

Comparison of methods for storing and accessing the English subset of the 2012 Google Books n-grams

	Sorted text	As distributed	ZS
Storage required (TB)	21.2	2.6	**1.5**
Bulk read running time (s/GB of output)			
1 CPU	7.9	**5.3**	20.1
8 CPUs	7.9	5.3	**2.6**
Find the 3-gram `this is fun`, year = 1955	380 ms[a]	48 min	**85 ms**[a]
Find all 3-grams matching `this is *`	750 ms[b]	61 min	**215 ms**[b]
Structured metadata	No	No	**Yes**
Corruption detection	None	Late	**Early**
Partial download suppport	**Yes**	No	**Yes**
Well known format	**Yes**	**Yes**	Not yet

Note. Smaller is better. **Bold** highlights best entry in each row. "Sorted text" assumes one file per n-gram length, lexicographically sorted, with binary search. "As distributed" indicates the files distributed by Google, where each gzipped text file contains all n-grams of a particular length which (more or less) share the same two-letter prefix (e.g., there is one `.gz` file containing all 3-grams beginning with the letters `th`). "ZS" is measured using the current defaults, which are LZMA compression (compression level `0e`), index block fan-out of 1024, and 384 KiB data blocks (before compression).

[a]The main cost of these search operations is disk seeks, which are difficult to accurately measure in the presence of caching. To estimate the total seek time we count the number of empty-cache disk seeks required and then multiply for 12 ms; for sorted text we assume that this is the entire cost, and for ZS we measure the required CPU time (25 ms) and add this to the estimated seek time.

[b]Estimated as a search[a] plus a bulk read of 50 MB with 8 CPUs.

traditional formats at all, but ZS can exploit the additional processor cores to decompress 7.8x faster – a near-linear speedup. This means that in a head-to-head comparison on common multi-core processors, ZS files with LZMA compression outperform ordinary gzipped files on both file size and decompression speed.

Of course ZS's support for parallel decompression is orthogonal to the choice of compression algorithm. If we use ZS with gzip's deflate algorithm, then compared to gzip we will see similar file sizes,[5] similar single-CPU decompression speeds, and even faster

[5]The ZS format's additional header, record/block framing, index blocks, and loss of compression efficiency caused by compressing each data block separately do introduce some additional storage overhead as compared to plain gzip, but tests suggest that this overhead is <0.5%.

multiple-CPU decompression speeds.

Notice also that on the particular server where we made our measurements, bulk reads of uncompressed files – which are bottlenecked by the speed at which data can be read from disk – are actually slower than bulk reads of compressed files – whose smaller file sizes result in reduced IO. This will of course vary depending on the compressibility of the data and the relative speed of the IO versus CPU subsystems on any particular machine, but the pattern we see here is probably representative of many current commodity computers using cheap hard-drives and multi-core CPUs.

Locating an individual record. As described above, ZS's tree-based index allows it to locate a record with far fewer disk seeks than are required by binary search, and this more than compensates for the additional CPU cost of decompression. In practice we often make multiple queries in succession, in which case caching will further improve ZS's speed. The distributed gzip files provide no way to locate a record except by performing a linear scan of all 3-grams in the `th` file, stopping when the desired record is located; in our example, such a scan turns out to be more than thirty thousand times slower than ZS's index-based search.

Locating a large number of records with a given prefix. Because ZS stores data records in lexicographically sorted order, records which share a prefix always appear next to each other in the file; therefore finding all recorded continuations of `this is...` can be accomplished by first querying the index to locate the first matching record, and then performing a bulk read. A similar strategy works for the sorted text files, but for reasons discussed above both the query and the bulk read are slower than for ZS. The distributed gzip files are unsorted, so the only way to find all 3-grams matching a pattern is to do an exhaustive scan of the file containing them; but even if they were sorted, this would speed up prefix searches by only a factor of two on average.

Of course for non-prefix searches (e.g., finding all 3-grams matching `* is *`), the ZS file index does not help; for all three of the storage methods described here the only way to

```
{
    "corpus": "google-books-eng-all-20120701",
    "subset": "3gram",
    "build-info": {
        "user": "njsmith",
        "host": "polypore.ucsd.edu",
        "version": "zs 0.10.0-dev",
        "time": "2014-05-15T19:18:48.774036Z"
    },
    "record-format": {
        "type": "separated-values",
        "separator": "\t",
        "column-types": [
            "utf8",
            "int",
            "int",
            "int"
        ],
        "column-names": [
            "ngram",
            "year",
            "match_count",
            "volume_count"
        ]
    }
}
```

Figure 3. An example of JSON-encoded metadata, used by a ZS file containing the 2012 Google Books `eng-all` 3-grams.

perform a query like this is to read the entire set of 3-grams, which takes hours rather than minutes or milliseconds. Queries like `this * fun` are an intermediate case: we can first do an efficient prefix search to find all 3-grams matching `this * *`, and then do an exhaustive scan over this subset to find the ones which match `this * fun`.

Structured metadata. Every ZS file contains a key/value map in the header that is encoded using JavaScript Object Notation (JSON) and can be used to store arbitrary user-defined metadata. An example is given in Fig. 3. The ZS format itself puts no restrictions on this metadata, except that it must be valid JSON; in the example, we record: (a) a unique identifier for the overall dataset and this file's portion of it, (b) some information about when and how the file was built, which `zs make` adds by default, (c) some information about how the data records in this file are formatted – that they contain tab-separated values, with some hints as to the types and names of the different fields. Since every dataset is different we do not attempt to specify a rigorous grammar of possible record types; if you are creating a ZS file and find this example to be a useful model for your own data then we hope you will follow it more or less closely, and that useful conventions will arise over time.

We strongly recommend that every ZS file contain at least enough information in the metadata header to uniquely identify the dataset and whatever processing pipeline was used (i.e., if you have two versions of the same dataset then it should be possible to determine which is which!). Other items you may wish to consider including are your contact information, bibliographic references for papers that users of this data might want to refer to or cite, any relevant DOIs, and whatever else seems relevant and useful. By placing this information directly into the ZS file, you can be confident that it will stay together with your data no matter where it goes.

Corruption detection and data integrity. Computer hardware and software is not trustworthy, especially over spans of terabytes and years. Hard drives and RAID cards sometimes silently corrupt the data entrusted to them; cosmic rays and other problems cause bits to flip while resting in RAM; the data you transmit on a network may not match the data received on the other end. This last possibility is particularly worrisome for a distribution format. Stone and Partridge (2000) measured a variety of different real-world networks, and estimated that even the most reliable network in their study introduced undetected corruption into one out of every ten billion packets transmitted. This may sound small, but it means that if a hundred people used this network to download the ZS dataset from Table 1, then on average ten of them would end up with corrupted files.

Traditional distribution formats provide only ad hoc and inadequate measures for handling such problems. Text files, of course, provide no automatic mechanism at all for detecting corruption. Standard compression formats like gzip or xz provide internal checksums, but these provide protection only in limited circumstances, because they are not checked until we reach the end of the file, after all the data has been decompressed, returned, and presumably submitted for further processing. If we stop reading early (as is assumed by the timings in Table 1), or if we forget to check the return code from our decompression routine, then we have no protection at all. Sometimes this protection is supplemented by additional checksums (e.g., MD5 sums) made available alongside the data files themselves; these checksums may even be computed automatically by data archival systems. It is not very effective engineering to design an integrity checking system with a theoretical failure rate of 2^{-128} and then depend on fallible humans to remember to check it manually; humans generally have somewhat higher failure rates than this. But even if they didn't, this approach would still be problematic: it provides no protection against corruption which occurs after the file has been downloaded and checked (e.g., due to a failing disk drive), and it provides no protection against corruption which occurs in between when the data was originally generated and when the checksum is computed. (Eliminating this gap entirely is, of course impossible, but it should be minimized as much as possible.) As a real world example of this problem, in the course of preparing this paper we discovered that one of the 2012 Google Books n-gram files posted on Google's website is, in fact, corrupt – decompressing this file causes gzip to eventually error out. Google's file distribution system automatically provides MD5 checksums of all files, and in this case the MD5 claims that the file is valid: the corruption appears to have occurred at some point in between when the file was compressed and when it was posted for download. The only thing that checking this MD5 actually tells us is that the file we have downloaded is corrupt in the same way as the file that they have stored.

ZS takes these dangers seriously. As a first layer of defense, the header and every block is protected by its own 64-bit CRC checksum. Together, these checksums protect every byte of the file. The use of multiple local checksums provides two major advantages: first, if corruption does occur, it makes it possible to determine which portions of the file are affected; this way we only have to re-download those portions, not the whole file, or –

in extremis – if all available copies are corrupted, then we can combine the uncorrupted portions from different copies. Second, it makes it possible to efficiently validate all – and only – the data that we access during any particular operation; unlike standard compression tools, we don't have to read the whole file just to check the first byte. This efficiency then makes it possible for the ZS software to provide a strong guarantee: we always validate data *before* returning it, so no corrupted data will ever be passed to downstream processing.[6] The cost of this checking turns out to be negligible: it is included in all of the times cited in Table 1. In addition, as a second layer of defense, each ZS file's header includes a SHA-256 hash of all the data records it contains. This hash is relatively slow to compute and validating it requires decompressing the entire file, but it provides an extremely strong test of data integrity. And, as a bonus, it provides a quick and easy mechanism to check whether or not two ZS files contain identical data, regardless of storage details like compression settings or index fan-out.

Because all of these checksums are integrated into the file format itself, they are guaranteed to be present and to stay with the file throughout its distribution lifecycle. This approach provides automatic, end-to-end integrity checking and avoids the problem we encountered above with the MD5 checksum generated by the distribution system.

Partial download support. Most modern web servers allow clients to download not just complete files, but arbitrary byte ranges. As shown in the examples section above, the ZS tools can take advantage of this feature to work directly with remote ZS files. To make this efficient, the block pointers stored inside the header and index blocks are careful to always note both the offset and length of the pointed-to block; this allows the pointed-to block to be retrieved with only a single round-trip to the server.

We envision two main use cases for this feature. First, on local networks, it provides an easy way to make a large dataset accessible to smaller computers: just place it on a web server in the lab. If your ping time to the server is less then 10 ms or so, then accessing a

[6]Technically, this is only a probabilistic guarantee; there is a 1-in-2^{64} chance that a 64-bit checksum will fail to detect accidental corruption. This is, for all practical purposes, indistinguishable from zero.

ZS file over the network should be essentially indistinguishable from accessing it on local disk. Second, it means that those who post large ZS datasets for public download will also automatically make them available for efficient querying by those who cannot afford to download the whole dataset. This may substantially increase the usability – and hence impact – of large datasets. For users, such queries will be noticeably slower than using local disk, because each each query requires multiple round trips to the server – e.g., with a ping time in the hundreds of milliseconds, the time to lookup a single entry will be measured in seconds. This is rather worse than the 85 ms quoted above for local disk access, but is still much faster, and uses fewer server resources, than downloading a terabyte of data! And in a way, this slowdown is reassuring: it means that light users can access the data directly, but that heavy users will be better off downloading the data once instead of being tempted to hammer on a public server.

Risks of relying on a novel format. The primary downside of the ZS format in comparison to the other formats considered here is that it is relatively new and unknown. This creates a risk that potential users may be unfamiliar with the format, or if files are stored in this format for long-term archival, that there will not be software available to read it in ten or a hundred years.

We have taken a number of measures to mitigate this risk. To encourage third-party implementations, the format itself is simple and fully documented – the Appendix contains not just a specification of the file format itself, but a bare-bones implementation of a decompressor in 49 lines of code (Fig. A2). The full-featured reference implementation is portable, undergoes exhaustive automated testing (>98% coverage) after every modification, and has been placed under a permissive BSD-style license, allowing it to be used freely in both open- and closed-source systems. And finally, to encourage interoperability between implementations, we have written an automatic file format validator (`zs validate`), which can be used to exhaustively verify that a ZS file conforms to the documented specification, and thus guarantee interoperability with any compliant reader software.

Conclusion

The advent of massive datasets such as the Google n-grams has created new scientific opportunities – but these datasets are extraordinarily difficult to work with using standard tools. The ZS format overcomes these problems: ZS files are smaller than the currently most popular compressed distribution formats, yet allow faster bulk and random access than uncompressed files. At the same time, they provide substantially better integrity checking than competitor formats, as well as advanced features like integrated metadata and the ability to remotely query any ZS file that has been made available for download over the web. Together these features make even the largest datasets accessible to ordinary researchers, and make ZS a compelling format for distributing, archiving, and using record-oriented datasets of any size.

Free software for creating and accessing ZS files is available at `http://vorpus.org/ZS`. The 2012 Google Books n-grams are available in ZS format at `http://cpl-data.ucsd.edu/zs/google-books-20120701/`.

Appendix

This appendix provides a complete specification of version **0.10** of the ZS file format, along with rationale for specific design choices.[7] 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. The most up-to-date version of this specification can be found at `http://vorpus.org/ZS`.

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

[7]In the final paper this will be replaced with the version 1.0 specification.

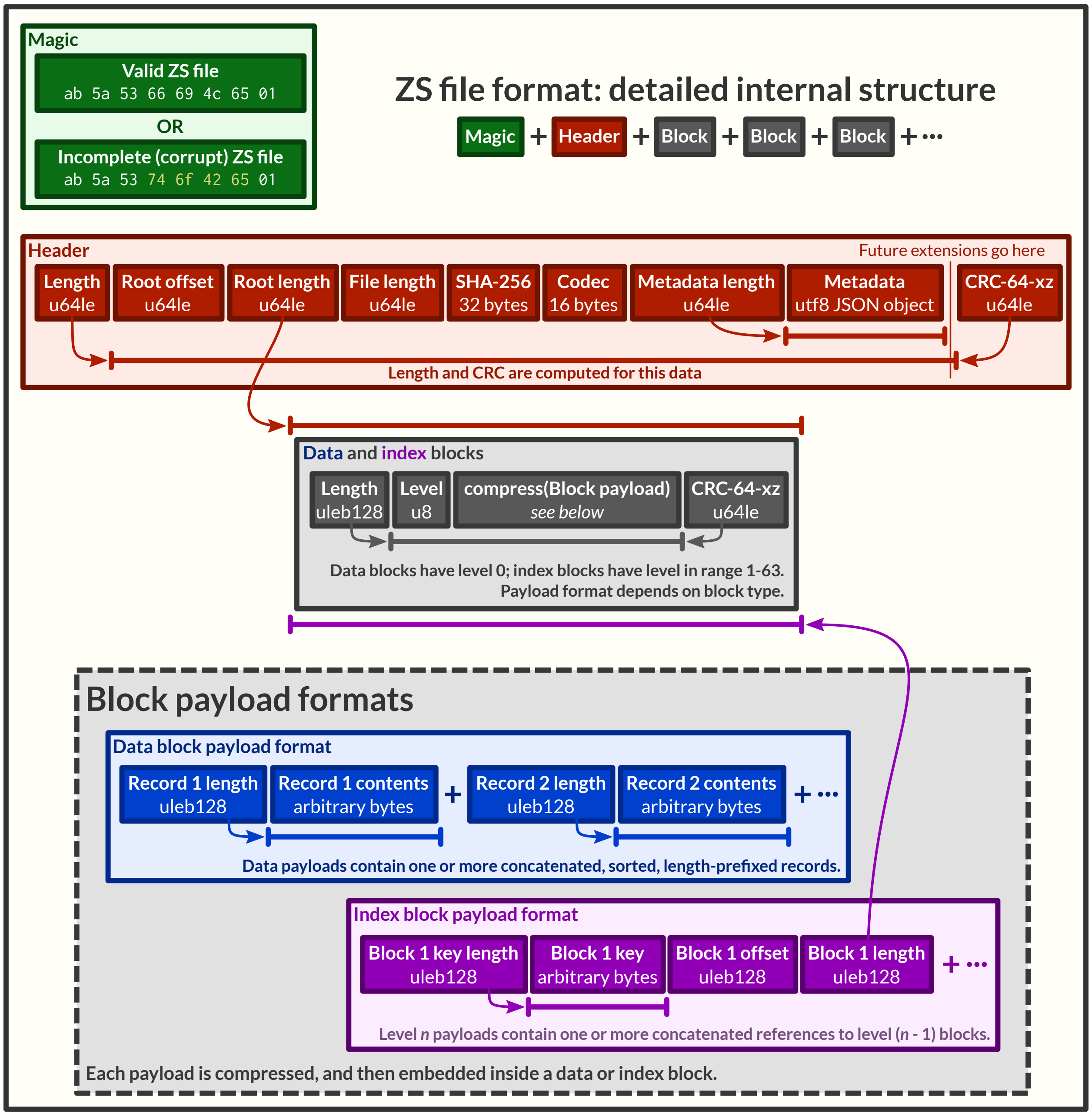
ZS file format: detailed internal structure
Magic + Header + Block + Block + Block + ···
Magic
Valid ZS file
ab 5a 53 66 69 4c 65 01
OR
Incomplete (corrupt) ZS file
ab 5a 53 74 6f 42 65 01
Header
Future extensions go here
Length u64le
Root offset u64le
Root length u64le
File length u64le
SHA-256 32 bytes
Codec 16 bytes
Metadata length u64le
Metadata utf8 JSON object
CRC-64-xz u64le
Length and CRC are computed for this data
Data and index blocks
Length uleb128
Level u8
compress(Block payload) see below
CRC-64-xz u64le
Data blocks have level 0; index blocks have level in range 1-63.
Payload format depends on block type.
Block payload formats
Data block payload format
Record 1 length uleb128
Record 1 contents arbitrary bytes
Record 2 length uleb128
Record 2 contents arbitrary bytes
+ ···
Data payloads contain one or more concatenated, sorted, length-prefixed records.
Index block payload format
Block 1 key length uleb128
Block 1 key arbitrary bytes
Block 1 offset uleb128
Block 1 length uleb128
+ ···
Level n payloads contain one or more concatenated references to level (n - 1) blocks.
Each payload is compressed, and then embedded inside a data or index block.

Figure A1. The disk layout of a ZS file.

```python
# Usage: python3 unzs.py input-file.zs output-file.txt
import sys, struct, zlib, lzma, io

def read_uleb128(f):
    value = shift = 0
    while True:
        byte_str = f.read(1)
        if not byte_str: # Check for end-of-file
            return None
        value |= (byte_str[0] & 0x7f) << shift
        shift += 7
        if not (byte_str[0] & 0x80):
            return value

def none_decompress(zpayload):
    return zpayload

def deflate_decompress(zpayload):
    return zlib.decompress(zpayload, -15) # -15 indicates raw deflate format

def lzma_decompress(zpayload):
    filters = [{"id": lzma.FILTER_LZMA2, "dict_size": 2 ** 20}]
    return lzma.decompress(zpayload, format=lzma.FORMAT_RAW, filters=filters)

decompressors = {b"none": none_decompress,
                 b"deflate": deflate_decompress,
                 b"lzma2;dsize=2^20": lzma_decompress}

in_f = open(sys.argv[1], "rb")
out_f = open(sys.argv[2], "wb")
in_f.seek(8)
header_length = struct.unpack("<Q", in_f.read(8))[0]
in_f.seek(72)
codec = in_f.read(16).rstrip(b"\x00")
decompress = decompressors[codec]
in_f.seek(8 + 8 + header_length + 8)
while True:
    block_len = read_uleb128(in_f)
    if block_len is None:
        break
    block = in_f.read(block_len)
    in_f.read(8) # skip CRC
    if block[0] == 0: # data block
        payload = io.BytesIO(decompress(block[1:]))
        while True:
            record_len = read_uleb128(payload)
            if record_len is None:
                break
            out_f.write(payload.read(record_len) + b"\n")
```

Figure A2. A minimal, self-contained, fully-compliant decompressor for ZS files, written in Python 3 (and tested with Python 3.3). In practice we recommend using a more full-featured implementation that includes proper error checking, but this code serves as executable documentation of how to get your data out in case of emergency.

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, as shown in Fig. 2, 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.

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 and disk accesses.

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 (Bradner, 1997).

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 (Collin & Pavlov, 2009). The Rocksoft model (Williams, 1993) 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 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. This is a popular variable-length format for storing unsigned integers of arbitrary size; it is also used by protocol buffers and the XZ file format, among others. The name comes from use in the DWARF debugging format, and stands for **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
```

In principle this format allows for redundant representations by adding leading zeros, e.g. the value 0 could also be written `80 00`. However, doing so in ZS files is forbidden; all values MUST be encoded in their shortest form.

Layout details

Figure A1 gives the picture of how a ZS file is laid out on disk – 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 (Deutsch, 1996a). Note that this is different from both the gzip format (Deutsch, 1996b) and the zlib format (Deutsch & Gailly, 1996), 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 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.
- <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, but does include the level field – see diagram. Note that this is calculated directly on the compressed disk representation of the block, *not* the decompressed payload.

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.

References

Arnon, I., & Snider, N. (2010). More than words: Frequency effects for multi-word phrases. *Journal of Memory and Language*, *62*(1), 67–82. doi: 10.1016/j.jml.2009.09.005

Baayen, R. H., Piepenbrock, R., & Gulikers, L. (1995). *CELEX2, LDC96L14.* Philadelphia, PA: Linguistic Data Consortium.

Balota, D. A., Yap, M. J., Hutchison, K. A., Cortese, M. J., Kessler, B., Loftis, B., ... Treiman, R. (2007). The English lexicon project. *Behavior Research Methods*, *39*(3), 445–459.

Bannard, C., & Matthews, D. (2008). Stored word sequences in language learning the effect of familiarity on children's repetition of four-word combinations. *Psychological Science*, *19*(3), 241–248. (PMID: 18315796) doi: 10.1111/j.1467-9280.2008.02075.x

Bansal, M., & Klein, D. (2011). Web-scale features for full-scale parsing. In *Proceedings of the 49th annual meeting of the Association for Computational Linguistics: Human Language Technologies-volume 1* (pp. 693–702). Association for Computational Linguistics.

Bradner, S. (1997, March). *Key words for use in RFCs to Indicate Requirement Levels* (No. 2119). RFC 2119 (Best Current Practice). IETF. Retrieved from `http://www.ietf.org/rfc/rfc2119.txt`

Brants, T., & Franz, A. (2006). *Web 1T 5-gram version 1, LDC2006T13.* Philadelphia, PA: Linguistic Data Consortium.

Brants, T., & Franz, A. (2009). *Web 1T 5-gram, 10 european languages version 1, LDC2006T13.* Philadelphia, PA: Linguistic Data Consortium.

Brysbaert, M., & New, B. (2009). Moving beyond Kučera and Francis: A critical evaluation of current word frequency norms and the introduction of a new and improved word frequency measure for American English. *Behavior Research Methods*, *41*, 977–990.

Carlson, A., & Fette, I. (2007). Memory-based context-sensitive spelling correction at web scale. In *Machine learning and applications, 2007. ICMLA 2007. sixth international conference on* (pp. 166–171). IEEE.

Collin, L. (n.d.). XZ utils [Computer program]. Retrieved from `http://tukaani.org/xz/`

Collin, L., & Pavlov, I. (2009). *The .xz file format.* Retrieved from `http://tukaani.org/xz/xz-file-format.txt`

Deutsch, P. (1996a, May). *DEFLATE Compressed Data Format Specification version 1.3* (No. 1951). RFC 1951 (Informational). IETF. Retrieved from `http://www.ietf.org/rfc/rfc1951.txt`

Deutsch, P. (1996b, May). *GZIP file format specification version 4.3* (No. 1952). RFC 1952 (Informational). IETF. Retrieved from `http://www.ietf.org/rfc/rfc1952.txt`

Deutsch, P., & Gailly, J.-L. (1996, May). *ZLIB Compressed Data Format Specification version 3.3* (No. 1950). RFC 1950 (Informational). IETF. Retrieved from `http://www.ietf.org/rfc/rfc1950.txt`

Evert, S. (2010). Google web 1T 5-grams made easy (but not for the computer). In *Proceedings of the NAACL HLT 2010 sixth web as corpus workshop* (pp. 32–40). Association for Computational Linguistics.

Flor, M. (2013, January). A fast and flexible architecture for very large word n-gram datasets. *Natural Language Engineering*, *19*(01), 61–93. doi: 10.1017/S1351324911000349

Germann, U., Joanis, E., & Larkin, S. (2009). Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Proceedings of the workshop on software engineering, testing, and quality assurance for natural language processing* (pp. 31–39). Stroudsburg, PA, USA: Association for Computational Linguistics.

Goldberg, Y., & Orwant, J. (2013). A dataset of syntactic-ngrams over time from a very large corpus of english books. In *Second joint conference on lexical and computational semantics (*SEM)* (Vol. 1, pp. 241–247).

Hawker, T., Gardiner, M., & Bennetts, A. (2007). Practical queries of a massive n-gram database. In *Proceedings of the Australasian language technology workshop* (pp. 40–48).

Heafield, K. (2011). KenLM: faster and smaller language model queries. In *Proceedings of the sixth workshop on statistical machine translation* (pp. 187–197). Stroudsburg, PA, USA: Association for Computational Linguistics.

Kaldager, S. V. (2012). Indexing google 1T for low-turnaround wildcarded frequency queries. In *Proceedings of the 2012 conference of the North American chapter of the Association for Computational Linguistics: Human Language Technologies: Student research workshop* (pp. 17–22). Association for Computational Linguistics.

Kučera, H., & Francis, W. N. (1967). *Computational analysis of present-day American English.* Providence, RI: Brown University Press.

Kudo, T., & Kazawa, H. (2009). *Japanese web n-gram version 1, LDC2009T08.* Philadelphia, PA: Linguistic Data Consortium.

Lin, D., Church, K. W., Ji, H., Sekine, S., Yarowsky, D., Bergsma, S., … Rao, V. (2010). New tools for web-scale n-grams. In *Proceedings of the seventh international conference on language resources and evaluation (LREC '10).*

Lin, Y., Michel, J.-B., Aiden, E. L., Orwant, J., Brockman, W., & Petrov, S. (2012). Syntactic annotations for the Google Books ngram corpus. In *Proceedings of the ACL 2012 system demonstrations* (pp. 169–174).

Liu, F., Yang, M., & Lin, D. (2010). *Chinese web 5-gram version 1, LDC2010T06.* Philadelphia, PA: Linguistic Data Consortium.

McDonald, S. A., & Shillcock, R. C. (2003a). Eye movements reveal the online computation of lexical probabilities during reading. *Psychological Science*, *14*(6), 648–652.

McDonald, S. A., & Shillcock, R. C. (2003b). Low-level predictive inference in reading: The influence of transitional probabilities on eye movements. *Vision Research*, *43*, 1735–1751.

Michel, J., Shen, Y. K., Aiden, A. P., Veres, A., Gray, M. K., Team, T. G. B., … Aiden,

E. L. (2011). Quantitative analysis of culture using millions of digitized books. *Science*, *331*(6014), 176–182. doi: 10.1126/science.1199644

Mitchell, T. M., Shinkareva, S. V., Carlson, A., Chang, K.-M., Malave, V. L., Mason, R. A., & Just, M. A. (2008). Predicting human brain activity associated with the meanings of nouns. *Science*, *320*(5880), 1191–1195. (PMID: 18511683) doi: 10.1126/science.1152876

Pauls, A., & Klein, D. (2011). Faster and smaller n-gram language models. In *Proceedings of the 49th annual meeting of the Association for Computational Linguistics: Human Language Technologies-volume 1* (pp. 258–267). Association for Computational Linguistics.

Piantadosi, S. T., Tily, H., & Gibson, E. (2011, March). Word lengths are optimized for efficient communication. *Proceedings of the National Academy of Sciences*, *108*(9), 3526 –3529. doi: 10.1073/pnas.1012551108

Sekine, S. (2008). A linguistic knowledge discovery tool: Very large ngram database search with arbitrary wildcards. In *Coling 2008: Companion volume – posters and demonstrations* (pp. 181–184). Association for Computational Linguistics.

Sekine, S., & Dalwani, K. (2010). Ngram search engine with patterns combining token, POS, chunk and NE information. In *Proceedings of the seventh international conference on language resources and evaluation (LREC '10).* Valletta, Malta.

Shaoul, C., Westbury, C., & Baayen, H. (2013). The subjective frequency of word n-grams. *Psihologija*, *46*(4), 497–537. doi: 10.2298/PSI1304497S

Smith, N. J., & Levy, R. (2011). Cloze but no cigar: The complex relationship between cloze, corpus, and subjective probabilities in language processing. In L. Carlson, C. Hoelscher, & T. F. Shipley (Eds.), *Proceedings of the 33rd annual conference of the Cognitive Science Society* (pp. 1637–1642). Boston, MA: Cognitive Science Society.

Smith, N. J., & Levy, R. (2013). The effect of word predictability on reading time is logarithmic. *Cognition*, *128*(3), 302–319.

Stone, J., & Partridge, C. (2000). When the CRC and TCP checksum disagree. In *ACM SIGCOMM computer communication review* (Vol. 30, pp. 309–319). ACM.

Tremblay, A., Derwing, B., Libben, G., & Westbury, C. (2011). Processing advantages of lexical bundles: Evidence from self-paced reading and sentence recall tasks. *Language Learning*, *61*(2), 569–613. doi: 10.1111/j.1467-9922.2010.00622.x

Williams, R. N. (1993). *A painless guide to CRC error detection algorithms (version 3).* Retrieved from `http://www.ross.net/crc/download/crc_v3.txt`

Wilson, M. D. (1988). The MRC psycholinguistic database: Machine Readable Dictionary, version 2. *Behavioural Research Methods, Instruments and Computers*, *20*(1), 6–11.