Google Books: Making the public domain universally accessible

Adam Langley and Dan S. Bloomberg

{agl,dbloomberg}@google.com

ABSTRACT

Google Book Search is working with libraries and publishers around the world to digitally scan books. Some of those works are now in the public domain and, in keeping with Google's mission to make all the world's information useful and universally accessible, we wish to allow users to download them all.

For users, it is important that the files are as small as possible and of printable quality. This means that a single codec for both text and images is impractical. We use PDF as a container for a mixture of JBIG2 and JPEG2000 images which are composed into a final set of pages.

We discuss both the implementation of an open source JBIG2 encoder, which we use to compress text data, and the design of the infrastructure needed to meet the technical, legal and user requirements of serving many scanned works. We also cover the lessons learnt about dealing with different PDF readers and how to write files that work on most of the readers, most of the time.

Keywords: Google, books, PDF, public domain, JBIG2, leptonica, Hausdorff, correlation, mixed raster, open source

1. INTRODUCTION

Google Book Search launched in late 2004 as part of Google's mission to organize the world's information and make it universally accessible and useful. Book Search consists of two programs: the publisher program and the library project. The first allows publishers to submit their books so that users can find them when searching. The second involves agreements with several major libraries, including Oxford's Bodleian and the University of Michigan, to scan and include their collections in our index. It is this latter project which results in our acquisition of works for which the copyright has expired worldwide, and hence we may freely share the results.

These books are available, in full, for viewing online at `http://books.google.com`, but this interface presents several challenges. First, the images are only 575 pixels wide and so the resolution is minimal for all but the very smallest books. Second, a user has to be online to read the book, frustrating those who might want to read on other devices such as e-book readers. And lastly, it's not practical to print the book out from the web interface.

Because of this we wanted to offer a format where users could read the book offline, in programs other than a browser, and which offered sufficient resolution for high quality printing.

1.1. PDFs - a widely supported mixed-raster container

We have little interest in writing client-side applications for the viewing of our contents. The breadth of different platforms in use means that such an undertaking would need to run in many diverse environments and, even if the project were open source, the effort required to support it is not practical.

Two widely used and standard formats for mixed-raster documents are PDF[1] and DJVU.[2] A decision of which to use depends on factors involving both the encoding and decoding processes. On the encoding side, we will see that the best tradeoff between coding rate and image quality is to have lossy compression for both foreground and background, and both PDF and DJVU support high quality lossy compression for images and text. However, DJVU encoding uses non-standard compression methods, for both the continuous tone background (wavelet) and the binary foreground mask, whereas PDF

uses encoding methods that have gone through standards committees for both (JPEG2000 and JBIG2, respectively). For decoding, the user's ability to view the pages depends on the availability of client programs to rasterize the page images. Both standalone and browser plug-in clients exist for DJVU and PDF, but the installed base for PDF clients, notably Adobe's free Acrobat Reader, is far larger. Additionally, utilities for printing PDF are more universal and efficient.

It is also important that the resulting files are as small as possible. The raw inputs to the compression process are several gigabytes in size and, even given the increasing penetration of broadband, few of our users would be willing to download and store such huge files. Thus we want to use the best compression formats so that the user experience is as good as possible. These formats are, inevitably, more costly in terms of computation for both the encoding and decoding. However, we have the computing resources to perform the encoding and are willing to invest them because of the importance of minimising file size. The computation involved in decoding was an unforeseen issue which will be covered later in the paper.

We were then presented with two decisions: which compression formats to use for images and which for text. Our major sources of images are book covers, drawings in the books and misclassified text. Because these books are out of copyright, they are old and have very few full-colour photos; most images in the books are black and white. The covers of books are often colour, but they carry little important information and it's not useful to have high quality representations of them. Misclassified text is a reality of dealing with these old books. It's rare, but we should be careful that the resulting text is still readable, even if of a much lower quality than correctly identified text.

We selected a number of representative samples of these three types of images and took the JPEG format as our baseline (with the IJG encoder). This was compared with the wavelet based JPEG2000 using the Kakadu encoder.[3] We reduced the JPEG quality level so that the image data was as small as possible, while still being of acceptable quality. Then we tuned the size of the JPEG2000 images until we felt they were roughly the same quality. This is a difficult comparison to make because the different formats degrade in different ways when stressed. In JPEG the edges of the DCT blocks become noticeable and mosquito noise appears where high frequency components cause ringing. In JPEG2000, texture disappears and areas smudge out. In this highly subjective test, JPEG2000 resulted in smaller images and this is what we chose to use.

JPEG2000 does not have a defined quality metric like that of JPEG, so we use a slope-distortion metric that is specific to the Kakadu encoder. In our tests, this appears to work well and it matches closely with our subjective impression of image quality.

In choosing a text compression format, our goal is to offer 600-dpi, 1-bit images at the lowest possible bitrate. This is a very respectable resolution and is easily readable on screen and when printed. Each page is about 4MB in size, uncompressed. For an example, see Figure 1.

In PDF, there are three possible candidates for the compression format: G4, FLATE and JBIG2. G4 is simple: it involves a fixed Huffman table which is used to encode the bits of the image as a raster. It is lossless, quick and each page is encoded independently. FLATE is also lossless, using a univerisal coder (zlib) along with 2D context estimation. However, for text the compression is not as good as G4.

We used the book Flatland[4] to test the efficacy of these different formats. FLATE compressing this 161 page book results in a 19.4MB PDF file (120KB/page), G4 produces 8.5MB (52 KB/page) but JBIG2* achieves 4.2MB (26KB/page).

JBIG2 is significantly more complex. It can optionally use Huffman compression, but in practice arithmetic encoding is used. It can encode the images in a raster way, like G4, but to get the best compression it can find symbols on the page, encode them and then encode the placement information. It can also work on each page independently, but to get the best compression rates multiple pages must be compressed at once.

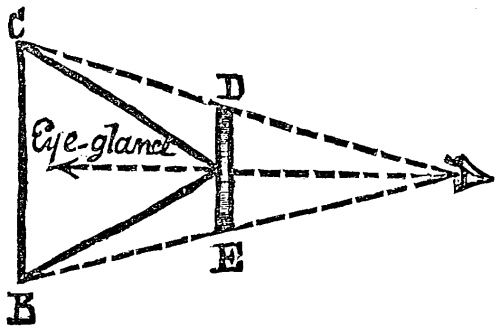
A brief survey of commercial encoders turned up none that appeared suitable. The encoder had to be able to run within our custom, Linux based environment. Further, past experience suggests that binary-only libraries cause future problems when ABIs change. Thus we embarked on writing our own JBIG2 encoder[5] for this project using a component classifier from the leptonica open source library. In the next section we describe the features of this classifer that make it suitable for use in a JBIG2 encoder, followed by a description of the JBIG2 encoder itself.

*With a context size of 20 pages

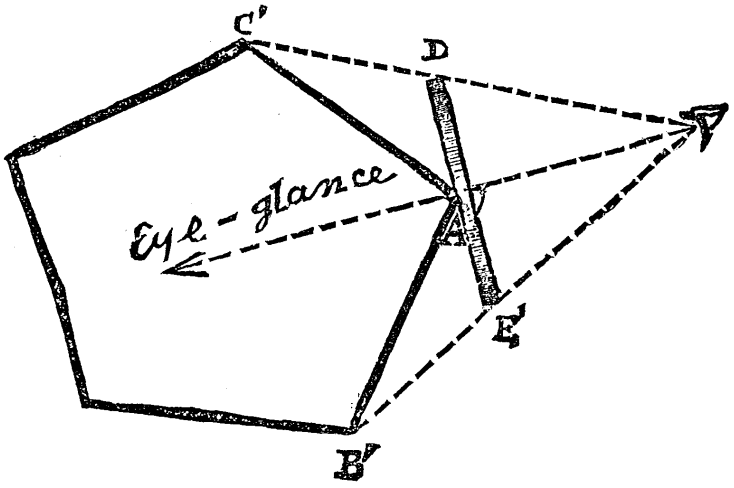
with great exactness the configuration of the object observed.

An instance will do more than a volume of generalities to make my meaning clear.

Suppose I see two individuals approaching whose rank I wish to ascertain. They are, we will suppose, a Merchant and a Physician, or in other words, an Equilateral Triangle and a Pentagon: how am I to distinguish them?



It will be obvious, to every child in Spaceland



who has touched the threshold of Geometrical Studies, that, if I can bring my eye so that its glance

Figure 1. An example of a 600-dpi page: page 44 of the Flatland PDF

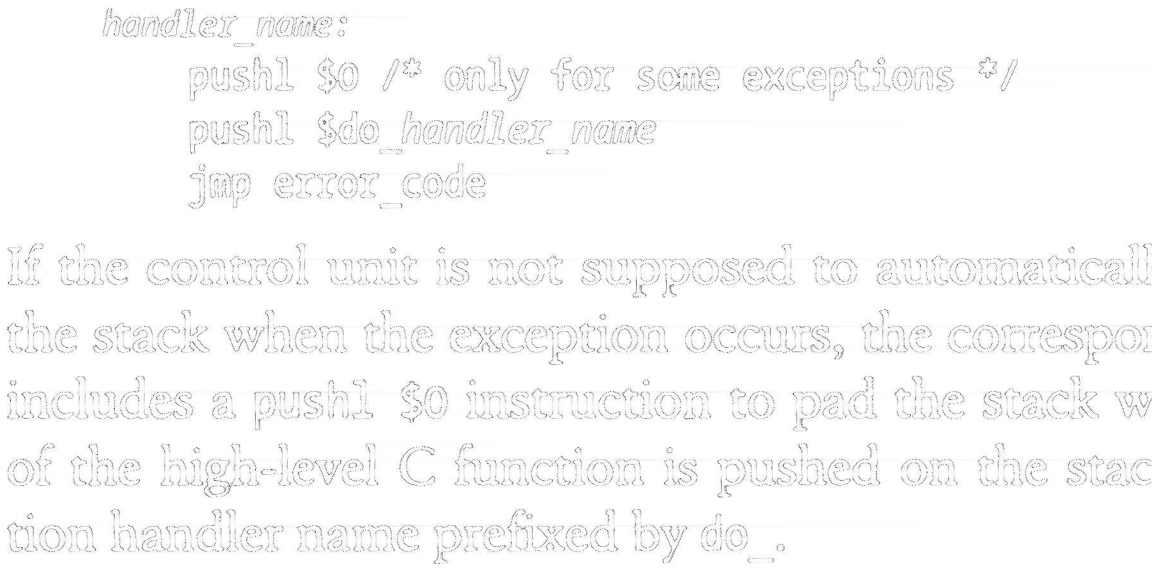


Figure 2. A JBIG2 arithmetic encoded generic region heatmap: the darker the pixel the more bits were spent encoding it. As expected, edges cost the most to code; horizontal and vertical edges are cheaper.

2. JBIG2 COMPRESSION

A JBIG2 bit stream can be thought of as a domain specific language for drawing pages and, as such, it has three basic modes of operation which we will be considering:

- *Generic region encoding*: an adaptive arithmetic encoding of 1 bpp (bit per pixel) images
- *Symbol encoding*: The ability to build dictionaries of 1 bpp images and place them on the page. The dictionaries can be shared across pages and a page may use several dictionaries
- *Refinement*: The ability to take a 1 bpp image and correct it by altering some of its pixels. The mask of pixels is arithmetically encoded and the unrefined image serves as context to the encoder.

There exist Huffman versions of all of these, and also a mode for encoding halftone regions. We do not support either in our encoder, nor do we consider them here. Huffman compression is inferior and halftone regions are classified as images and JPEG2000 encoded.

The most obvious way to compress pages is to losslessly encode them as a single 1 bpp image (see Figure 2). However, we get much better compression by using symbol encoding and accepting some loss of image data. Although our compression is lossy it is not clear how much information is actually being lost - the letter forms on a page are obviously supposed to be uniform in shape, the variation comes from printing errors and lack of resolution in image capture.

Given that we have chosen to perform symbol encoding, the encoder is as simple as possible. Connected components are gathered from a set of some number of pages, those which appear on more than one page are placed in a global symbol dictionary, and the rest are placed in a per-page dictionary. The page information itself consists of placement information for the symbols. Refinement is not used since it is costly to correct too many pixels and it makes little difference visually.

2.1. Unsupervised classification of binary components

The leptonica implementation of the JBIG2 classifier[6] performs a greedy unsupervised classification of 'tokens' in a 1 bpp image. The 'tokens' can be connected components (CC) or aggregates of them. The method is to make a decision, for every new token, whether it belongs to an existing class or should form a template for a new class. The challenge is to provide

an implementation that works efficiently for millions of tokens, but also has sufficient flexibility to allow controlling the accuracy of the match within a class. As such, our implementation has a number of useful and important features, each of which is considered in more detail:

- On the input side, it will classify as tokens either CC, characters, or words in the roman alphabet.
- The image comparison function can be either a windowed rank Hausdorff or a windowed correlation. The correlation method has a correction factor depending on the weight (thickness) of the characters.
- For each new instance, it does the matching over a small subset of existing templates that expands sub-linearly with the number of templates.
- To get the best template location for each instance, a final correlation-based correction is performed after the centroids are aligned.

2.1.1. Selection of binary tokens for classification

Connected components are found efficiently using Heckbert's run-based flood fill[7] to successively remove each one, computing the bounding box at the same time. Characters typically require aggregation of components including dots, and are extracted from a mask generated by a morphological closing of the input image with a vertical structuring element (*Sel*). Words are trickier, because the spacing between words can vary significantly, and they are extracted at a resolution of about 150 ppi (pixels/inch). The interword spacing is analyzed statistically by carrying out a sequence of horizontal dilations with a Sel of width 2, each dilation followed by a computation of the number of CC. With successive dilations, the number of CC falls as the characters within a word have been joined. This is then followed by a plateau in the CC count, where the words remain separated, and finally, with further dilations, the words begin to join and the number of CC resumes its fall.

We next consider two methods, Hausdorff and correlation, for comparing two image "tokens" to make a decision whether they are sufficiently similar to be assigned to the same class. In all cases, one of the tokens is a template for an existing class, and the other is an instance of a component to be tested.

2.1.2. Hausdorff image comparator

The Hausdorff distance H is a true metric (that obeys the triangle inequality) for comparing two 1 bpp images.[8] It is defined as the maximum of two directed Hausdorff distances, h, where the directed Hausdorff distance between images A and B is the maximum over all pixels in A of the distance from that pixel to the closest pixel in B. Formally, if we define the distance from a point p in A to the nearest point in the set B to be $d(p, B)$, then the directed Hausdorff distance from A to B is

$$h(A, B) = \max_{(p \in A)} d(p, B)$$

and the Hausdorff metric is

$$H(A, B) = \max(h(A, B), h(B, A))$$

The Hausdorff distance is typically sensitive to pixels that are some distance away from the the nearest boundary pixel. Because one or both images can have salt or pepper noise that is far from a boundary pixel, a rank Hausdorff comparator,[9,10] with a rank fraction slightly less than 1.0 can be used to give some immunity to such noise. However, rather than actually computing the Hausdorff distance between the two tokens, which is expensive, a decision is simply made whether the distance is less than some threshold value. Additionally, the comparison is made for a single relative alignment where the two tokens have coincident centroids. With these simplifications, the template and instance can be dilated in advance. Then the Hausdorff test checks that the dilated image of one contains all (or a rank fraction) the pixels of the undilated image of the other, and the tests are run in both directions. A successful match causes assignment of the instance to the template's class.

The strength of the Hausdorff match is that it uses pixels far from any boundary, where variability in pixel value is expected to be low. For an odd dimension Sel, say 3×3, with the origin of the Sel at the Sel center, dilation is symmetric, and the Hausdorff distance threshold is an integer (1 for 3×3, 2 for 5×5, etc.). However, for small text, character confusion can occur even using a Hausdorff distance threshold of 1. Furthermore, for very small components, such as punctuation, using a 3×3 Sel in the presence of halftone dots often results in use of irregular halftone dot components as templates for

punctuation. Therefore, it is necessary to use a 2×2 Sel, which is asymmetric about its origin, to get acceptable results. It is then necessary to choose a rank fraction of about 0.97. Use of 0.99 or greater gives far too many classes, and use of 0.95 or smaller results in character confusion.

2.1.3. Correlation image comparator

Because very tiny Hausdorff distance thresholds are required to correctly classify small text components, the pixels near the boundary are important. Consequently, correlation comparators are preferred to rank Hausdorff, because the former can be more finely tuned. When using a correlation comparator, the centroids are again aligned when doing the comparison. Let image tokens 1 and 2 be the instance and template, respectively. Denote the number of foreground pixels in images 1 and 2 by $|1|$ and $|2|$, and the number in the intersection of the two images as $|1 \oplus 2|$. Then the correlation is defined to be the ratio: $(|1 \oplus 2|)^2/(|1| \times |2|)$. This score is compared with an input threshold. However, because two different thick characters can differ in a relatively small number of pixels, the threshold is modified depending on the fractional foreground occupancy of the template, given by $R = |2|/(w_2 \times h_2)$, where w_2 and h_2 are the template dimensions. The modified threshold t' is related to the input threshold t and a weighting parameter f ($0.0 \leq f < 1.0$) by: $t' = t + (1.0 - t) \times R \times f$. Values of t = 0.8 and f = 0.6 form a reasonable compromise between accuracy and number of classes for characters with 300 ppi scans.

2.1.4. Hashing for efficient encoding

The classifier is typically used on text characters, and it must be designed to avoid putting different characters in the same class. Because there is relatively little penalty for oversegmenting (i.e., having multiple classes for the same typographic character), a large document consisting of hundreds of pages can have thousands of templates. The classifier must efficiently match each input component (instance) against this expanding set of templates. The number of resulting templates is reduced by removing the halftone image regions before classifying the remaining components. Even more important, the time to classify an instance should grow significantly slower than the number of templates. We only need to consider templates that are roughly the same size as the instance, but we can not afford a linear search through the set of templates. To find the possible templates efficiently, hash the size into one of a large set of buckets (we use the prime number 5507) by taking (width × height) mod 5507. Then set up an array (`NumaHash`) of arrays of template ids for each bucket. Because the instance dimensions can differ from its template by up to 2 pixels, for each instance, list the 25 buckets given by ((width ± 0,1,2) × (height ± 0,1,2)) mod 5507 and match the templates in these buckets against the instance, starting at the center and spiralling out. For each template in this ordered list, first check the actual dimensions, because we hash on the product. Then measure the similarity (correlation or rank Hausdorf) in a greedy fashion: (1) when the number of non-conforming pixels exceeds the allowed rank value, go to the next template; (2) accept the first template where the match completes within the allowed rank value.

2.1.5. Final correlation to reduce baseline jitter

After an instance is matched to a template, it is necessary to find the best location. Surprisingly, although the matching score is computed with centroids aligned, in a significant fraction of instances, the best alignment (correlation-wise) differs by one pixel from centroid alignment. This correction is important for appearance of text, where the eye is sensitive to baseline wobble due to a one-pixel vertical error. We find the nine correlation scores between template and instance, allowing the template location to move up to one pixel in both horizontal and vertical directions from centroid alignment. The position chosen is that with a minimum in the number of pixels in the XOR between the two images.

3. BUILDING THE PDF

3.1. Linearisation

Linearised PDFs (see appendix F of the PDF specification[1]) are a special organisation of PDFs designed for incremental loading over the web. They appear well formed to readers that don't understand the linearisation. However, for readers that do understand it, which is currently limited to Adobe's browser plugin, there are several structures that allow the first page to be displayed immediately and further pages to be displayed without downloading the whole.

The objects needed for the first page appear at the beginning of the file, along with a hint table that gives the offsets, lengths and other information for the rest of the pages. A special dictionary is also included at the beginning of the file that marks the file as being linearised. Upon seeing this, a capable reader can close the HTTP connection to stop the download

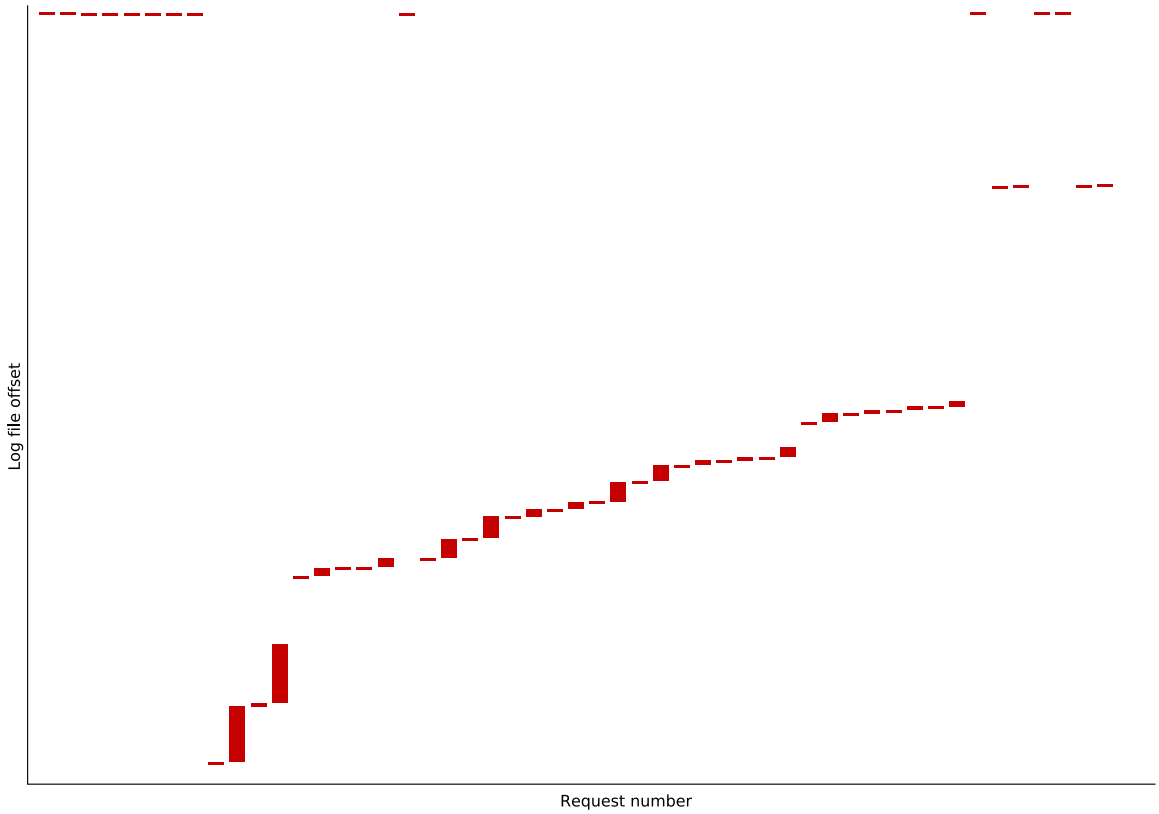


Figure 3. A graphical representation of the requests made by the Acrobat Plugin when opening a linearised PDF file

and then issue requests for specific byte ranges on a new connection. To reduce latency, the Adobe plugin also preloads some number of pages in advance.

This would appear to be ideal for us, reducing both the bandwidth and the user's latency. However, we do *not* linearise our files due to several factors. The first is the unfortunate behavior of the plugin. Figure 3 shows the byte ranges requested by the plugin following the opening of a linearised version of one of our PDF files. The vertical axis is the logarithm of the byte offset and the horizontal axis indexes the range requests. The taller the bar, the larger the range requested, but this is complicated by logarithmic y axis which causes requests closer to the beginning of the file to appear larger. Requests which ended up exceptionally short in this graph where expanded to a minimum size.

From the graph one can see that the plugin starts by loading from the cross-reference section at the end of the PDF and then loads the hint tables and objects of the first page. It also preloads several following pages, generating many more requests.

These range requests are batched into HTTP requests in groups of about four. Some ranges are contiguous (and yet not merged) and some ranges even overlap. Even when ranges are very close together (separated by as little as one byte), separate ranges are requested. If we were to serve these requests, they would be load balanced across many different servers and each must perform a significant amount of processing in order to serve a tiny reply. This amount of overhead is unacceptably wasteful.

Second, generating the linearised PDFs adds a lot of complexity. As the above data shows, we did implement this, but it is very difficult using only the Adobe plugin as an oracle of correctness; the PDF specification contains 11 implementation notes documenting ways in which their own appendix is incorrect. Also, we must consider the cost of maintaining this code. Complexity in code costs forever more.

The existing web interface already provides a way for users to read a small section of the book, and the ability to download PDFs is designed for those who are downloading the whole book to print it or read offline. Because linearisation

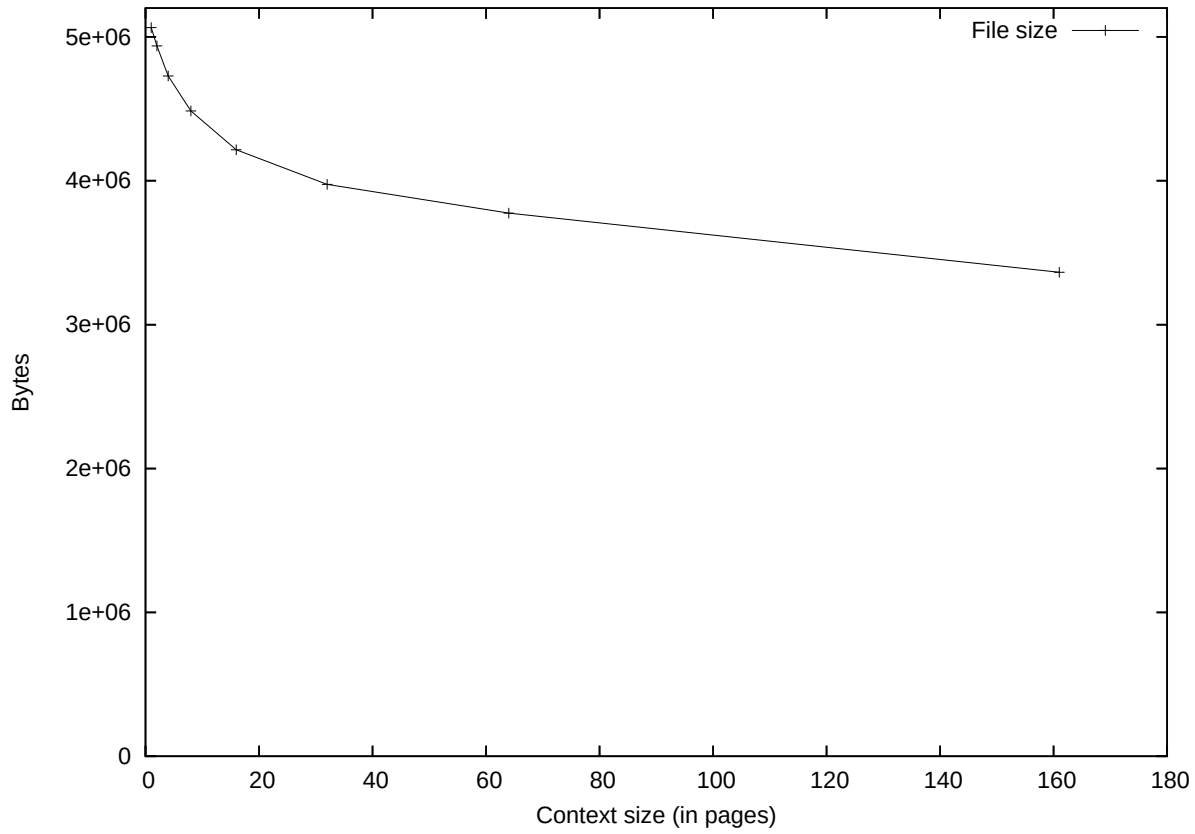


Figure 4. The file size (in bytes) of Flatland with different compression context sizes.

offers little advantage in this mode of use, and considering as the issues outlined above, we chose not to linearise these files.

3.2. JBIG2 context size

With an image format that can extract cross-page information, it is beneficial to present as many pages as possible to the compressor. Figure 4 shows the file size of the resulting PDF when different numbers of pages are presented to the JBIG2 compressor at once. In this example the book is Flatland,[4] which has 161 pages. This shows clearly the improvement with increasing context size. A large context size will slow down the compression, but we have the computational resources to compress entire books at once, and because the smaller file sizes would benefit our users we would choose to do so.

Unfortunately, both our canonical PDF viewers (`xpdf` and Acrobat Reader) slow down considerably as the context size increases. To measure this we instrumented `xpdf` to time the rendering process. We had `xpdf` render the first 20 pages of Flatland with varying context sizes. Figure 5 shows the results. The user experience (in terms of time to render a page) degrades markedly as context size increases; the experience with Acrobat is similar.

Because `xpdf` is open source we can tell why this happens: `xpdf` fails to cache the large, shared symbol dictionary and renders it once for every page. We can't look at the Acrobat Reader source, but it seems very likely that the issue is the same. We could have fixed `xpdf`, but not Acrobat and so to decision was made to use a context size of 16 in production. Hopefully, at some point in the future the readers will be more capable. Then, larger context sizes should actually speed up the rendering time, at the cost of extra initial processing and memory use, and we will be able to shrink all the files.

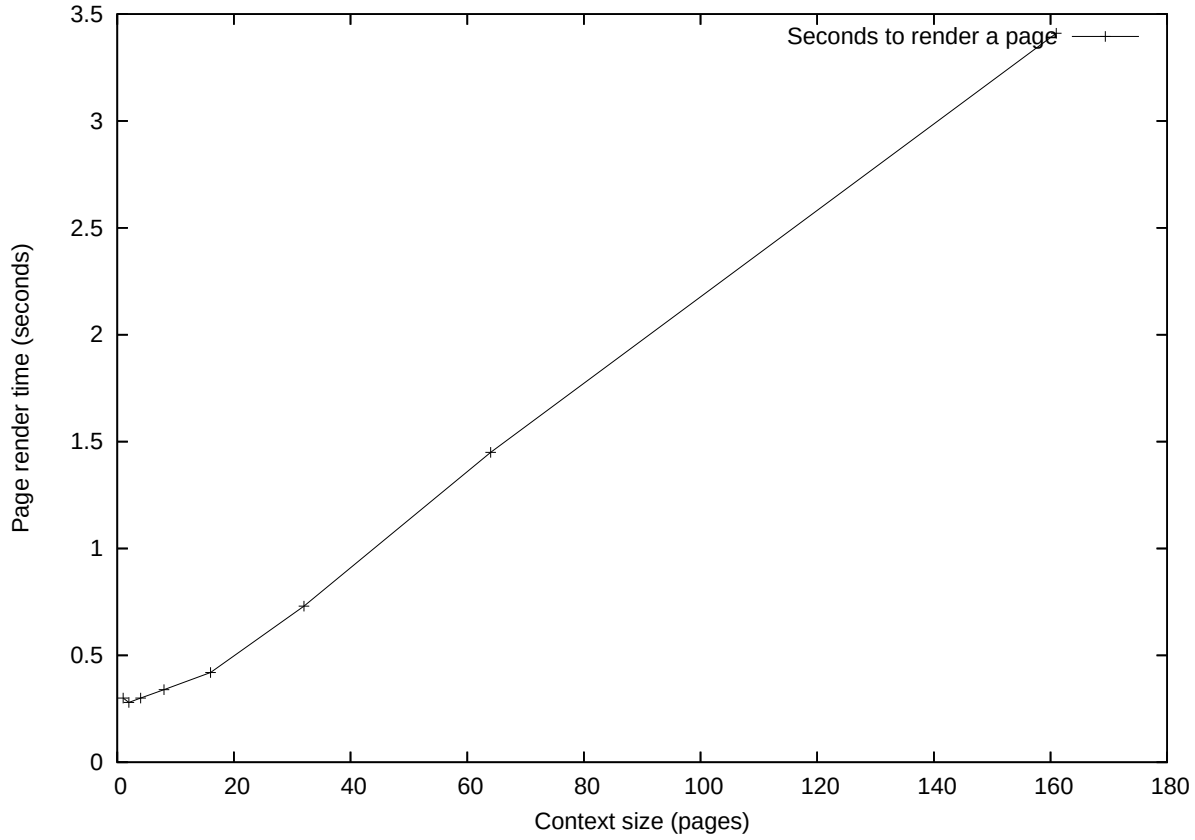


Figure 5. The time (in seconds) for `xpdf` to render a page of Flatland with different compression context sizes.

4. SERVING INFRASTRUCTURE

Once the PDFs are generated, they must be served to the world at large. This is complicated by the fact that the exact number of pages and the contents of at least the first two of these pages is not fixed until we have the HTTP request for the file.

At the beginning of every PDF file are one or more pages of legal text, and these need to be internationalised to the language of the user. Because these files can be copied between users once they are out in the wild, we include the legal text first in the language of the user (as determined by the language of the web interface that they are using) and then in the language of the book, the hope being that anyone who reads the book can, at least, read the second of these.

If the two languages are the same, we only include one copy of the legal text, thus the number of pages in the PDF can change with each request. This means that we cannot simply generate PDFs and serve them with a web server, nor can we generate the PDFs from raw data on the fly due to the amount of computation involved. Thus, generating the PDFs is a two stage process. First the raw data for each book is processed into an intermediate form and a number of byte strings, which are stored. Second, at serving time those byte strings are fetched on the fly and built into a final PDF file.

The intermediate form contains a number of page structures, one for each non-dynamic page in the book. Each page contains an optional JBIG2 stream, an optional JBIG2 context stream (shared by several pages) and zero or more JPEG2000 streams with position information. Each page structure contains everything except the image bitstreams themselves, which are referred by their number. A set of such page structures is called the outline of the book. The outline contains all the information needed to determine the byte offset of every object in the PDF.

The raw book data is stored in GFS[11] (Google File System) at several locations. The processing of the raw data to generate the intermediate form involves all the image compression work and hence is very compute intensive. This

computation is carried out off-line in a process similar to a MapReduce.[12] The raw data for each book is mapped to a set of image bitstreams and the outline data for the book. Using the resources at Google, the whole process for all books takes less than two days to complete.

The outlines are stored in a Bigtable.[13] This is a sparse, distributed, persistent multi-dimensional sorted map, developed at Google that maps tuples of (row, column, time) to a string. In this case, the row is our book identifier, the columns contain the elements in the outline, and we ask for the most recent entry.

The byte strings are also stored in a separate Bigtable where the row is "bookid/streamnumber" and there is a single data column. Again, we ask for the most recent entry. Because of the way Bigtable works, this groups the byte strings for a given book together as much as possible, which improves the hit rates of the caches.

Bigtable only provides for transactions at the row level, so we need a way to prevent the regeneration of the intermediate form from partially overwriting live data which is being served to the world. So, before we start writing, we find the greatest index of any byte string in the Bigtable. When new byte strings are added they are numbered starting from this index, so that they don't overwrite any live data. Finally, the new outline is inserted (atomically), which makes the new data live. After a day we can be sure that any downloads that were reading the old data have ended, so the old data can be removed.

We can now describe the serving process. When a request for a PDF arrives, we fetch the outline from the Bigtable. Based on this, the language of the book and the language of the user, we compute the length of the PDF and the locations where all the byte strings need to be inserted. This allows us to return HTTP reply headers that include the `Content-Length` header. From here the non-image regions of the PDF are generated on demand and we start pipelining fetches for byte strings from the Bigtable, interleaving them as required.

5. ACKNOWLEDGMENTS

We have given a technical description of a scalable system and some of the components we built for serving, quickly and efficiently, a very large number of scanned books over the internet. However, the description is incomplete without highlighting a very unusual aspect of the context in which the work was done; namely, through a publicly held corporation (Google). As mentioned in the beginning, the Book Search program fits clearly into Google's mission to make information accessable and useful. In pursuit of this goal, Google is maximizing the accessibity of books that are in the public domain, both through online search and serving, and for offline reading and printing. And there is no charge to the user for the services. We are very grateful to a large number of people for supporting this vision and implementing these policies; in particular, Product Managers Adam Smith and Adam Mathes, and Engineering Directors Dan Clancy and Chris Uhlik.

Additionally, Google internally makes tremendous use of Open Source software, and supports Open Source development in a number of ways. We wish to thank Chris DiBona for developing a number of Google Open Source initiatives, and for being a proactive advocate of open sourcing Google code.

REFERENCES

1. *PDF Reference*, Adobe System Incorporated, fifth ed.
2. "Djvu." `http://www.djvuzone.org`.
3. "Kakadu." `http://www.kakadusoftware.com`.
4. E. A. Abbott, *Flatland, A Romance of Many Dimentions*, Little, Brown, and Company, 1899.
5. A. Langley, "`jbig2enc`." `http://www.imperialviolet.org/jbig2.html`.
6. D. S. Bloomberg, "*Leptonica*." `http://www.leptonica.org/jbig2.html`.
7. A. S. Glassner, ed., *Graphics Gems*, Academic Press, 1990. 275–277,721–722.
8. D. Huttenlocher, D. Klanderman, and W. Rucklidge, "Comparing images using the Hausdorff distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **15**, pp. 850–863, September 1993.
9. D. S. Bloomberg and L. Vincent, "Blur hit-miss transform and its use in document image pattern detection," *SPIE Conf. 2422, Doc. Rec. II* , pp. 278–292, Feburary 1995.
10. D. S. Bloomberg and L. Vincent, "Pattern matching using the blur hit-miss transform," *Journal Elect. Imaging* **9(2)**, pp. 140–150, April 2000.
11. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *19th ACM Symposium on Operating Systems Principles*, pp. 29–43, 2003.

12. J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *OSDI’04: Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, 2004.
13. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, and et al, “Bigtable: A distributed storage system for structured data,” in *OSDI’06: Seventh Symposium on Operating System Design and Implementation*, pp. 725–726, 2006.