Compressed Perfect Embedded Skip Lists for Quick Inverted-Index Lookups*

Paolo Boldi
Sebastiano Vigna
Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano
(Draft)

Abstract

Large inverted indices are by now common in the construction of web-scale search engines. For faster access, inverted indices are indexed internally so that it is possible to skip quickly over unnecessary documents. The classical approach to skipping dictates that a skip should be positioned every $\sqrt{f}$ document pointers, where f is the overall number of documents where the term appears. We argue that due to the growing size of the web more refined techniques are necessary, and describe how to embed a *compressed perfect skip list* in an inverted list. We provide statistical models that explain the empirical distribution of the skip data we observe in our experiments, and use them to devise good compression techniques that allow us to limit the waste in space, so that the resulting data structure increases the overall index size by just a few percents, still making it possible to index pointers with a rather fine granularity.

1 Introduction

Inverted indices are one of the most commonly used techniques to organise very large (web-scale) document collections. They provide high-speed access to sets of documents satisfying queries, which can be subsequently ranked and returned to the user. If properly built, they are also extremely compact [11].

The birth of web search engines has brought new challenges to traditional inverted index techniques. In particular, *eager* (or *term-at-a-time*) query evaluation has been replaced by *lazy* (or *document-at-a-time* [10]) query evaluation. In the first case, the inverted list of one of the terms of the query is computed first (usually, choosing the rarest term [11]), and then, it is merged or filtered with the other lists. When evaluation is lazy, instead, inverted lists are scanned in parallel, retrieving in sequence each document satisfying the query. The latter approach is essential in very large document collections, where the actual number of documents that could be retrieved is guessed (usually with a statistical approach), documents are reordered using some kind of static ranking, and the scan for documents satisfying the query is stopped as soon as enough documents have been retrieved.

Lazy evaluation requires keeping constantly in sync several inverted lists. To perform this operation efficiently, it is essential that a *skip* method is available that allows the caller to quickly reach the first document pointer larger than or equal to a given one. The classical solution to this problem [6] is that of embedding skipping information in the inverted list

*[**This work has been partially supported by a "Finanziamento per grandi e mega attrezzature scientifiche" of the Università degli Studi di Milano.**]

itself: at regular intervals, some additional information describe a *skip*, that is, a pair given by a document pointer and the number of bits that must be skipped to reach that pointer. The analysis of skipping given in [6] concludes that skips should be spaced as $\sqrt{f_t}$, where f_t is the frequency of term t (i.e., the number of documents containing t), and that one level of skip is sufficient for most purposes.

Nonetheless, the abovementioned analysis has two important limitations. First of all, it does not contemplate the presence of *positions*, e.g., of a description of the exact position of each occurrence of a term in a document (something which is necessary to evaluate proximity), or of application-dependent additional data: as a result, the estimate of the cost of *not* skipping a document record is severely underestimated; second, it is fundamentally based on eager evaluation, and its conclusions cannot be extended to lazy evaluation.

Not only lazy evaluation is necessary for easier ranking (especially with the long and complex queries that arise in automated query preprocessing): new techniques applied to inverted indices, such as *document sampling* [1], require that skips are generated by a random source. In that case, it is even more difficult to predict the usage pattern of skip methods.

Motivated by these reasons, we are going to present a generic method to self-index inverted lists with a very fine level of granularity. Our method does not make any assumption on the structure of a document record, or on the usage pattern of the inverted index. Skips to a given pointer (or by a given amount of pointers) can always be performed with a logarithmic number of low-level reads and bit-level skips: nontheless, the size of the index grows just by a few percents, thanks to a sophisticated analysis of the skip structure that eliminates redundant information, and predicts with high precision the remaining data, so that compression techniques can be used to further reduce the space occupied by the skip structure.

Our techniques are particularly useful for *in-memory indices*, that is, for indices kept in core memory (as it happens, for instance, in Google), where most of the computational cost of retrieving document is scanning and decoding inverted lists (as opposed to disk access), and at the same time a good compression ratio is essential.

All results described in this paper have been implemented in MG4J, a full-text indexing engine that is available as free software at `http://mg4j.dsi.unimi.it/`.

2 Perfect Embedded Skip Lists for Inverted Indices

Inverted indices. Inverted indices are a basic tool for querying textual document collections [11, 3]. Consider a collection of N documents, where documents are numbered from 0 to $N-1$; each document is seen as a sequence of term occurrences. An *inverted index* for the collection is a data structure made of one *inverted list* for each term. The inverted list for term t contains one item (sometimes called a *posting*) for each document where t appears, in increasing order of document; every item contains the *(document) pointer* p (i.e., the number of the document), plus possibly some additional data, such as

- the *count*, that is, the number of occurrences of term t in document p;
- the *positions*, that is, the list of integers representing the positions where the term t appears in document p.

The number of items contained in the inverted index for term t (i.e., the number of documents where t occurs) is called the *frequency* f_t of t, and it is usually written at the beginning of the inverted list. A graphical example is shown in Figure 1.

Inverted lists are accessed sequentially, as their purpose is exactly to retrieve a sequence of documents containing a term; usually, when trying to match a query, several inverted lists

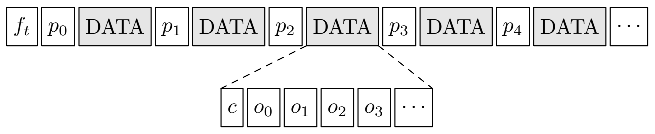

Figure 1: An inverted list without skips.

are read and merged. Introducing skips in these lists, as suggested in [6], may allow one to save time because in conjunctive queries one can skip over useless postings.

When a skip structure is embedded in an inverted list, we can tell how quickly a skip can be performed by counting the number of record read, the number of integer read, and the number of *jumps*, that is, low-level (usually, filesystem-level) calls that skip over a given number of bits. In the rest of this paper, we shall describe a data structure that is the natural evolution of the one-level skip described in [6], and show how this structure can be compressed and used efficiently.

Perfect skip lists. Skip lists [9] are a data structure in which elements are organised as in an ordered list, but with additional references[1] that allow one to skip forward in the list as needed. More precisely, a *skip list* is a singly linked list of items $\langle x_0, x_1, \ldots, x_{n-1}\rangle$, such that:

- items in the list are increasingly ordered w.r.t. some fixed order relation, that is, $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$;
- every item x_i appearing in the list contains a reference to the next item x_{i+1} (as in any linked list);
- moreover, item x_i contains a certain number $h_i \geq 0$ of extra references, that are called the *skip tower* of the item; the t-th reference in this tower addresses the first item $j > i$ such that $h_j \geq t$.

When searching a skip list for an item x, we scan the first tower from the top, stopping at the first reference pointing at an item smaller than or equal to x; then we follow the reference and restart the process until x is found, the current item is larger than x or no more items are available.

Skip lists are a probabilistic structure (albeit deterministic versions do exist [7]): a parameter $0 < p < 1$ is fixed, and then the height of each tower is chosen by tossing a p-biased coin until the outcome is positive. This probabilistic construction ensures that the structure can be updated dynamically, still maintaining logarithmic access time on average. An additional fixed upper bound avoids excessively tall towers.

We are now going to describe *perfect skip lists*, a deterministic version of skip lists that is suitable for inverted lists. Much like probabilistic skip lists can be thought of as a way to represent a search tree, a perfect skip list resembles a *complete* binary search tree; the big advantage of this representation is that it can be easily embedded into an inverted list with a high compression ratio, and that it is apt to sequential scanning.

For sake of simplicity, we start by describing an ideal, infinite version of a perfect skip list. Let $\mathrm{LSB}(x)$ be defined as the least significant bit of x if x is a positive integer, and ∞ if $x = 0$. Then, define the height of the skip tower of item x_i as $h_i = \mathrm{LSB}(i)$. In this idealised version, the tower of the first element has infinite height, as it must be able to skip

[1]The term reference used here can be taken to mean "memory address"; in the data structure and programming literature this is usually called *pointer*. Unfortunately, in the papers on inverted indices, pointer is used for "document number", and we shall adhere to this tradition.

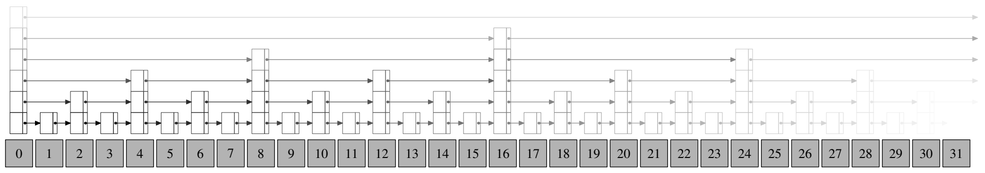

Figure 2: An infinite perfect skip list; the first tower is really infinite.

arbitrarily forward. Note that by choosing heights in this way, we can find any element in a logarithmic (in the list size) number of steps.

We now fix two limiting parameters: the number of items in the list, and the maximum height of a tower. Both parameters are fundamental in an actual implementation. We say that a finite skip list is *perfect* w.r.t a given height h and size T when:

1. no tower contains more than $h + 1$ references;
2. all references that would exist in the infinite perfect skip list are present, provided that they refer to an item with index smaller than or equal to[2] T, and that they do not violate the first requirement.

Theorem 1 In a perfect skip list with T items and maximum height h, the height of a tower at element i is

$$\min(h, \mathrm{LSB}(k), \mathrm{MSB}(T - i)) + 1$$

where $k = i \bmod 2^h$, and $\mathrm{MSB}(x)$ is the most significant bit of $x > 0$, or -1 if $x = 0$. In particular, if $i < T - T \bmod 2^h$ the tower has height

$$\min(h, \mathrm{LSB}(k)) + 1,$$

whereas if $i \geq T - T \bmod 2^h$ the tower has height

$$\min(\mathrm{LSB}(k), \mathrm{MSB}(T \bmod 2^h - k)) + 1.$$

Proof. Let $i = t2^h + k$, with $0 \leq k < 2^h$. The requirements for a perfect skip list claim that item i has a reference of level s only if

$$\begin{cases} s \leq h \\ i + 2^s \leq T \\ s \leq \mathrm{LSB}(i). \end{cases}$$

The second inequality is easily turned into $s \leq \mathrm{MSB}(T - i)$. We notice that $\mathrm{LSB}(i) = \mathrm{LSB}(t2^h + k)$: thus, when $k = 0$, $\mathrm{LSB}(i) \geq h$, because $t2^h$ has at least h trailing zeroes in its binary expansion; for the same reason, $\mathrm{LSB}(i) = \mathrm{LSB}(k)$ when $k > 0$ (as $k < 2^h$). We conclude that $\min(\mathrm{LSB}(i), h) = \min(\mathrm{LSB}(k), h)$, hence the first claim.

Let $T = v2^h + r$ with $0 \leq r < 2^h$. If $i < T - T \bmod 2^h$, then $T - i > r$, so $T - i = (v - t)2^h + r - k > r$. We conclude that $v > t$, so $T - i \geq 2^h$ and $\mathrm{MSB}(T - i) \geq h$. Finally, if $i \geq T - T \bmod 2^h$ then $v = t$, so $\mathrm{MSB}(T - i) = \mathrm{MSB}(r - k) < h$. ∎

A tower at i whose height is less than $\mathrm{LSB}(i) + 1$ will be called *truncated*, because its height is strictly smaller than the height it would have in the infinite list; in particular, the tower at 0 is always truncated.

[2]This means that occasionally there might be a reference pointing just after the end of the list.

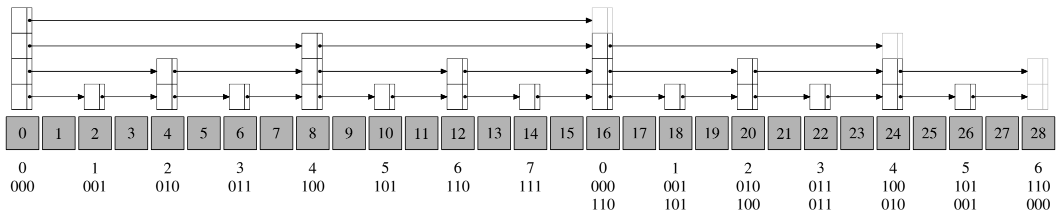

Figure 3: A perfect skip list with $T = 29$ items, $q = 2$ and $h = 3$; the ghosted references do not exist (they are truncated by minimisation with $\mathrm{MSB}(\lfloor L/q \rfloor - k)$). The first line show the values of k; the second line their h-bit binary expansion; for the defective block, also the binary expansion of $\lfloor L/q \rfloor - k$ is shown.

Addressing directly all pointers in an inverted list would create unmanageable indices. Thus, we shall index only one item out of q, where q is a fixed *quantum* that represents the minimally addressable block of items. Summarising, a *perfect skip list* with quantum q and maximum height h can be described as follows:

- items appearing in the list are logically grouped from left to right into *blocks* of $B = q2^h$ elements; the last block may contain less than B elements, in which case it will be called *defective*;
- within each block, only items whose index is a multiple of q will have a non-empty skip tower;
- an item appearing in position kq (where $k = 0, 1, \dots, 2^h - 1$) within its *non-defective* block contains a skip tower of height $\bar{h} = \min\{\, h, \mathrm{LSB}(k)\,\} + 1$: the s-th reference in this tower ($s = 0, 1, \dots \bar{h} - 1$) addresses the item appearing $q2^s$ items ahead (this is always an item in the same block, or the first item in the next block, or a virtual item appearing immediately after the end of the list);
- for a defective block with L items, the height of the skip tower at kq (where $k = 0, 1, \dots, \lfloor (L-1)/q \rfloor$) is $\min\{\, \mathrm{LSB}(k), \mathrm{MSB}(\lfloor L/q \rfloor - k)\,\} + 1$; note that, in particular, if after kq there are less than q items there will be no tower at all.

Figure 3 shows a perfect skip list with $T = 29$ items, $q = 2$ and $h = 3$; in this case every block contains $B = q2^h = 16$ items, so the example shows one non-defective block (items 0–15) and one defective block (items 16–28).

2.1 Embedding skip lists into inverted indices

The first problem that we have to deal with when trying to embed skip lists into an inverted index is that we want to access data in a strictly sequential manner, so the search algorithm we described cannot be adopted directly: we must store not only the bit offset of the referenced item, but also the pointer contained therein.

In Figure 4 the reader can see a portion of an inverted list with skips; in this example, $q = 3$, $h = 2$ and the figure shows a single block, containing exactly $q \cdot 2^h = 12$ item, plus the beginning of the next block. Every item is made by a document pointer p_0, p_1, … and the data section, which is represented as a light grey small rectangle. The skip tower, if any, appears between the document pointer and the data section, and it is dark grey in the figure.

Tower entries are written from the top, and every tower with more than one entry starts with an indication, in bits, of the length of the forthcoming tower (L_0, L_2 etc.). Every entry refers to an item appearing further on in the list, and contains

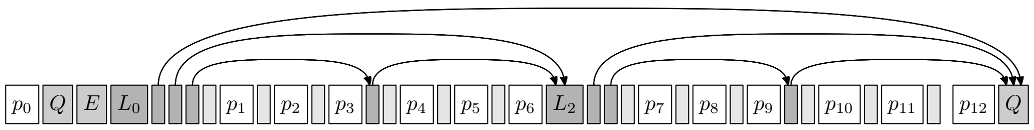

Figure 4: One block in an inverted list with skips ($q = 3$, $h = 2$).

- the *pointer skip*, that is, the gap w.r.t. the referenced pointer;
- the *bit skip*, that is, the number of bits that should be skipped to move to the referenced pointer (more precisely, the number of bits from the end of the tower to the point just after the referenced pointer).

At the beginning of each block, we write two extra items: the *quantum bit length* Q, that is, the average length of a quantum in bits in the forthcoming block (excluding the space occupied by the skip towers), and the *entry bit length* E, that is, the average length of a tower entry in bits in the forthcoming block.

Since tower entries are written starting from the top, we can in principle replicate exactly the standard skip list scanning algorithm. When searching for p, we have a *climb-up* phase in which we reach the highest tower pointing before p; a *block-jump* phase in which we skip entire blocks; and finally a *climb-down* phase in which we reach the quantum possibly containing p.

Both the climb-up and the climb-down phases require at most q record reads, $3\bar{h}$ integers reads and $\bar{h}$ jumps, where $\bar{h}$ is the height of the tallest tower in the list (since we know the tower lengths, we can read only that part of the tower we are actually interested in); the block-jump phase requires at most $5\lfloor f_t/B\rfloor$ integer reads and $\lfloor f_t/B\rfloor$ jumps. In particular, if $B \geq \max_t f_t$, no block jump phase will occur: in that case, the number of record read is bounded by $2q$, the number of integer read by $6\log(f_t/q)$ and the number of jumps by $2\log(f_t/q)$.

3 Representing and Compressing Skip Lists

Even with a reasonable quantum (e.g., $q = 64$), a perfect skip list requires a significant amount of space to be stored in raw form. In this section we show how to reduce this space drastically.

Pointer skips. Let us consider a document collection of N documents, where each term t appears with frequency (number of documents in which the term appears) f_t and relative frequency $p_t = f_t/N$; according to the *Bernoulli model* [11], every term t is considered to appear independently in each document with probability p_t. As a result, the random variable G_t representing a *gap* between document pointers, that is, the difference between *consecutive* document pointers containing the term t, is distributed as follows

$$\Pr[G_t = x] = \begin{cases} p_t(1-p_t)^{x-1} & \text{for } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

In other words, $G_t - 1$ is a random variable with geometric distribution. Let us now define a *pointer skip of distance ℓ for term t* as the difference between a pointer containing the term t and a pointer appearing ℓ position later in the inverted list, and denote the random variable representing a pointer skip of distance ℓ for term t by $S_{t,\ell}$. Clearly, $S_{t,\ell}$ is the sum of ℓ i.i.d. random variables G_t, so $S_{t,\ell} - \ell$ has negative binomial distribution with parameters

p_t and ℓ. Since we are interested in reasonably large values of ℓ, we will approximate it with a normal distribution[3] with mean ℓ/p_t and standard deviation $\sqrt{\ell(1-p_t)}/p_t$ (which are exactly the mean and standard deviation of $S_{t,\ell}$). In other words:

$$S_{t,\ell} \sim \Phi\left(\ell/p_t, \sqrt{\ell(1-p_t)}/p_t\right)$$

where $\Phi(\mu, \sigma)$ represents, as usual, a normal distribution with parameters μ and σ. A first approach could just store the difference with the mean (which we expect to be distributed normally around 0) using some suitable universal code. Experiments show, however, that there are a few problems with this approach. A first problem is that the Bernoulli model is a good model, but not a perfect one: in particular, actual document collections may sport correlation between adjacent documents. Moreover, the lack of truncation (a geometric distribution has a nonnegative probability for *any* positive integer, but no gap may actually be larger than N) tends to make the empirical mean slightly smaller than the theoretical mean. The effect on gaps is not so noticeable, but it becomes very visible on long-distance skips. On the contrary, the theoretical and empirical variances agree very well.

To minimise these inaccuracies, we suggest to predict the top pointer skip of a tower using the Bernoulli model, but to predict the remaining pointer skips using a *halving model*, in which a pointer skip of level s is stored as the difference from the skip of level $s+1$ that contains it, divided by 2.

How will the differences be distributed? If we were just comparing two arbitrary skips of different level, the two variables would be independent, and the resulting random variable would be $S_{t,q2^{s+1}}/2 - S_{t,q2^s}$; but this is not the case, as the larger skip actually comprises the smaller skip, so the actual distribution is

$$\left(S_{t,q2^s} + \bar{S}_{t,q2^s}\right)/2 - S_{t,q2^s} = S_{t,q2^s}/2 - \bar{S}_{t,q2^s}/2,$$

where $\bar{S}_{t,q2^s}$ is distributed as $S_{t,q2^s}$. The resulting mean is 0, as in the independent case, but the resulting variance, $\sqrt{q2^s(1-p_t)/2}/p_t$, is smaller by a factor of $\sqrt{3}$.

Gaussian Golomb Codes. We are now left with the problem of coding integers normally distributed around 0. Contrarily to what happens for geometric and double-sided geometric distributions [4, 5], there is no simple, instantaneous code for the normal distribution. Thus, when faced with the task of encoding skips, we must resort to some approximation (Huffman coding is, of course, out of question). More precisely, we shall compute approximately the best Golomb code for a given normal distribution. This is not, of course, an optimal code for the distribution, but we shall see that we are not losing much, and that the correct parameter for the Golomb code can be approximated easily in closed form.

Recall that $x \geq 0$ is encoded by a *Golomb code of modulus b* as $\lfloor x/b \rfloor$ in unary (i.e., $\lfloor x/b \rfloor$ zeroes followed by a one) followed by a minimal binary coding of $x \bmod b$ [11]. Since we need to code numbers from $\mathbf{Z}$, we shall map them through $\nu : \mathbf{Z} \to \mathbf{N}$, where

$$\nu(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ 2|x| - 1 & \text{otherwise.} \end{cases}$$

Let us define $\gamma(x) = e^{-x^2/(2\sigma^2)}$ and recall the definition of the error function

$$\operatorname{erf}(x) = (2/\sqrt{\pi}) \int_0^x e^{t^2}\, dt.$$

[3]In this application of the Central Limit Theorem, when we say that the discrete variable X is approximated with a normal distribution Φ we mean that $\Pr[X = x] \approx \int_{x-1/2}^{x+1/2} \Phi(t)\, dt$.

σ	1	4	7	10	13	16	19	22	25
b^* avg. length	2.43	4.16	4.94	5.48	5.84	6.14	6.40	6.62	6.79
loss vs. optimal b	12.64%	0.54%	0.%	0.10%	0.04%	0.17%	0.05%	0.%	0.05%
loss vs. entropy	18.56%	2.78%	1.70%	2.14%	1.62%	1.53%	1.75%	1.69%	1.51%
gain w.r.t. δ coding	16.91%	25.84%	27.61%	27.03%	26.91%	26.49%	25.89%	25.58%	25.45%

Table 1: A comparison of average lengths of Golomb codes for various normal distributions.

We are going to approximate the expected length of a Golomb coding of modulus b applied to such integers as

$$\int_{-\infty}^{-\frac{1}{2}} \frac{1}{\sqrt{2\pi}\sigma}\gamma(x)\ell_b(-2[x]-1)\,dx + \int_{-\frac{1}{2}}^{\infty} \frac{1}{\sqrt{2\pi}\sigma}\gamma(x)\ell_b(2[x]))\,dx$$

where $\ell_b(-)$ represents the number of bits required to encode a given natural number using a Golomb code with modulus b, and $[-]$ denotes the rounding function. Dropping the rounding, and approximating $\ell_b(x)$ with $x/b + \log b + 1$, we have to minimise a sum of integrals:

$$\int_{-\infty}^{-\frac{1}{2}} \gamma(x)\left(\frac{-2x-1}{b} + \log b + 1\right)dx + \int_{-\frac{1}{2}}^{\infty} \gamma(x)\left(\frac{2x}{b} + \log b + 1\right)dx.$$

Now, differentiating w.r.t. b we get

$$\int_{-\infty}^{-\frac{1}{2}} \gamma(x)\left(\frac{2x}{b^2} + \frac{1}{b\ln 2} + \frac{1}{b^2}\right) + \int_{-\frac{1}{2}}^{\infty} \gamma(x)\left(-\frac{2x}{b^2} + \frac{1}{b\ln 2}\right)dx =$$
$$\frac{2}{b^2}\left(\int_{-\infty}^{-\frac{1}{2}} \gamma(x)x\,dx - \int_{-\frac{1}{2}}^{\infty} \gamma(x)x\,dx\right) + \frac{1}{b\ln 2}\int_{-\infty}^{\infty}\gamma(x)\,dx + \frac{1}{b^2}\int_{-\infty}^{-\frac{1}{2}}\gamma(x)\,dx =$$
$$-\frac{4\sigma^2 e^{-1/(8\sigma^2)}}{b^2} + \frac{\sqrt{2\pi}\sigma}{b\ln 2} + \sqrt{\frac{\pi}{2}}\frac{\sigma}{b^2}\left(1 - \operatorname{erf}\left(\frac{\sqrt{2}}{4\sigma}\right)\right),$$

and value that makes the last expressions zero is

$$\bar{b} = 2\ln 2\sqrt{\frac{2}{\pi}}\sigma e^{-1/(8\sigma^2)} + \frac{\ln 2}{2}\left(\operatorname{erf}\left(\frac{\sqrt{2}}{4\sigma}\right) - 1\right).$$

However, ignoring the multiplicative factor $e^{-1/(8\sigma^2)}$ and the second summand yields the much simpler (and almost equivalent, at least when $\sigma \gg 1$)

$$b^* = 2\ln 2\sqrt{\frac{2}{\pi}}\sigma \approx 1.106\,\sigma.$$

Figure 5 shows the two predictions (that, as the reader can verify, are practically indistinguishable when $\sigma \gg 1$) against the optimal value computed experimentally. In Table 1 we show an empirical comparison of the average length produced by b^*.

Bit skips. The strategy we follow for bit skips is absolutely analogous to that of pointers, but with an important difference: it is very difficult to model correctly the distribution of bit skips. Remember that we are considering indices containing arbitrary data after the document pointers (e.g., the occurrence list): this means that the number of bit skips may depend on several variables, such as the number of occurrences of the term in each document.

It is possible that in some specific situations (e.g., very regular document collections, or indices with very predictable data blocks) such an analysis can be carried out with success,

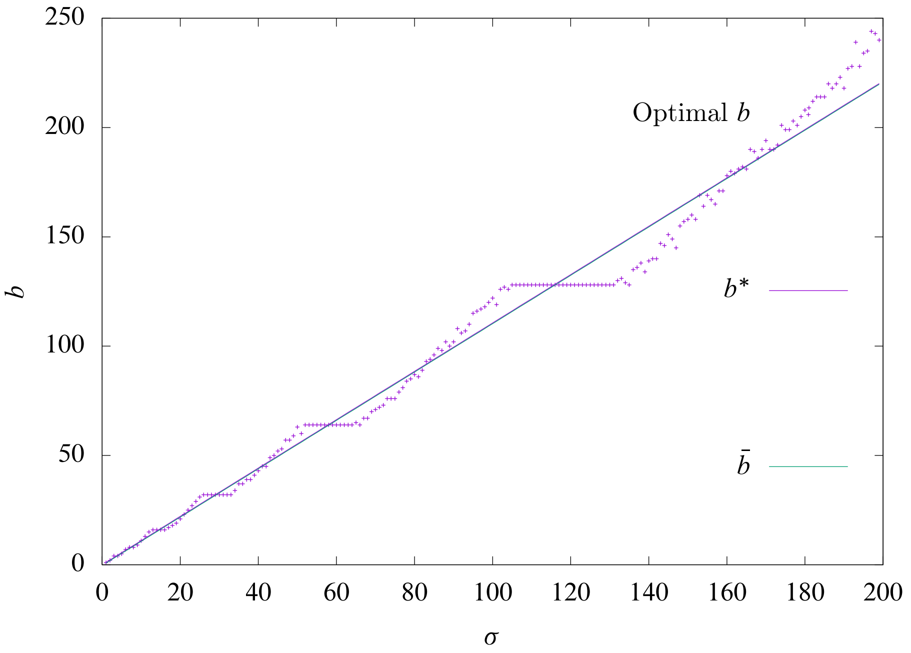

Figure 5: The optimal value of the Golomb modulus b for integers distributed as $\Phi(0, \sigma)$ while varying σ (computed experimentally) plotted against analytic predictions.

and we leave it for future work. Presently, however, we suggest that a preview scheme similar to that of document pointers, coupled with a universal code such that γ or δ, is a viable solution.

More precisely, as explained in Section 2.1, at the beginning of each block the index contains the number Q of bits consumed on the average by every quantum in the block, and the number E of bits consumed on the average by every tower entry.

A skip of level s will skip over 2^s quanta, hence it will occupy $2^s Q$ bits on the average, plus the space occupied by the skip entries. Since the bit skip specifies the number of bits *from the end of the current tower*, after the entry of level s we will have a whole tower with s entries, 2 towers with $s - 1$ entries, 4 towers with $s - 2$ entries etc.; all in all, we will have $2^{s+1} - s - 2$ entries, occupying about $(2^{s+1} - s - 2)E$ bits.

Summarising, the expected number of bits for a skip of level s is $2^s Q + (2^{s+1} - s - 2)E$, and top skips are stored as a difference from this quantity, using a δ code. For skips of lower levels, we use the same technique adopted for pointers: a bit skip of level s can be predicted from the bit skip of level $s + 1$ reducing the latter by $(s + 1)E$ and dividing the result by 2, because $\left(2^{s+1} Q + (2^{s+2} - (s + 1) - 2)E - (s + 1)E\right)/2 = 2^s Q + (2^s - s - 2)E$.

4 Inherited Towers

As remarked in the previous sections, the part of a tower that has greater variance (and thus is more difficult to compress) is the tower top. However this might seem strange, our next goal is to avoid writing tower tops at all.

The usage pattern of an inverted list is a sequence of reads and skips that start from the first document pointer, and possibly reach the end of the list. Perfect skip lists accelerate this process, but introduce more data than it is actually necessary. Indeed, if we assume that we shall always read an inverted list from the start (and this is necessary, if we want to compress document pointers), it is possible to maintain an *inherited tower* that represent all "skip knowledge" gathered so far. For instance, the top entry of the first tower of a block remains valid for the whole block length, and it points to the next block: we have just to update the entry as we move across the block. This idea resembles that of *search fingers* [8] for skip lists, but we have to adapt it because we do not assume direct access and hence cannot skip backwards in the list.

Whenever we read a tower entry we keep that entry (updating its pointer and bit skips) until we read another entry at the same level (see Figure 6). As a result, the climb-up phase is no longer necessary, as it is replaced by a scan of the inherited tower. This actually halves the number of jumps due to climbing.

Note that inherited entries might not reach height h if the block is defective (see the right half of Figure 3). Supposing without loss of generality that $q = 1$ (since inherited towers for positions in the middle of a quantum are obviously identical to those available at the start of the quantum), we have:

Theorem 2 The highest valid entry in an inherited tower for a defective block of length L is

$$\mathrm{MSB}(L \oplus k),$$

where $\oplus$ denotes bit-by-bit exclusive or and k is the offset into the defective block.

Proof. Assume w.l.o.g. $q = 1$. If an inherited entry at level $t \leq h$ exists in position x, it must have been inherited by the last tower containing an entry at level t, that is, by the tower positioned at $y = x - x \bmod 2^t$. Such an entry actually exists only if $h(y)$ is larger than or equal to t, that is, if

$$\min\{\, h, \mathrm{LSB}(y), \mathrm{MSB}(T - y) \,\} \geq t.$$

For non-defective blocks, the above inequality always holds, as $\mathrm{MSB}(T - y) \geq \mathrm{LSB}(y) \geq t$. Otherwise, there might be entries that are not inherited because they do not exist: this happen when $\mathrm{MSB}(T - y) = \mathrm{MSB}(L - x \bmod 2^h + x \bmod 2^t) < t$, where L is the length of the block. For instance, in Figure 3 the inherited tower at position 26 would not contain an entry at level 2 because $\mathrm{MSB}(13 - 12 + 2) < 2$.

What we need is an analytic form for the highest inherited entry available in defective blocks. Letting $k = x \bmod 2^h$, the inequation conditioning the existence of an inherited entry at level t becomes $L - k + k \bmod 2^t = L - \lfloor k/2^t \rfloor 2^t \geq 2^t$. This easily leads to $\lfloor L/2^t \rfloor > \lfloor k/2^t \rfloor$, which is true when

$$t \leq \mathrm{MSB}(L \oplus k),$$

where $\oplus$ denotes bit-by-bit exclusive or. ▮

It is a pleasant fact that this number is always h if $L = 2^h$, which means that we can use it to bound the height of the inherited tower even in non-defective blocks.

Where does a (inherited) tower entry point to? As shown in Figure 6, it might happen that consecutive entries refer to *the same item*. More precisely, the t-th level and the s-th level of the tower (including the inherited entries) at x refer to the same item iff $x + 2^t - x \bmod 2^t = x + 2^s - x \bmod 2^s$, that is, $2^t - 2^s = x \bmod 2^t - x \bmod 2^s$ (note that the $x + 2^t - x \bmod 2^t$ is a correct estimate of the pointed item also for non-inherited entries, as in that case $x \bmod 2^t = 0$). This happens exactly when all the bits of x between the t-th and the s-th are 1's.

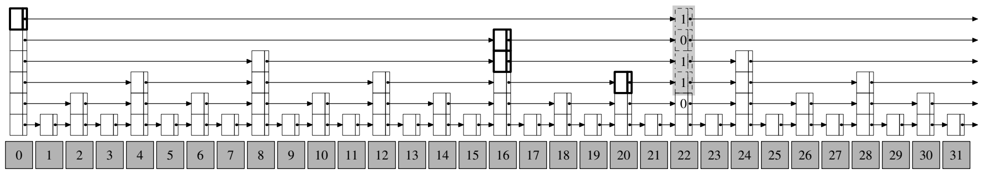

Figure 6: Scanning the list ($q = 1$, $h = 5$), we are currently positioned on element $x = 22$, with a tower of height 2; its inherited tower is represented in grey, and the items it is inheriting from are in bold.

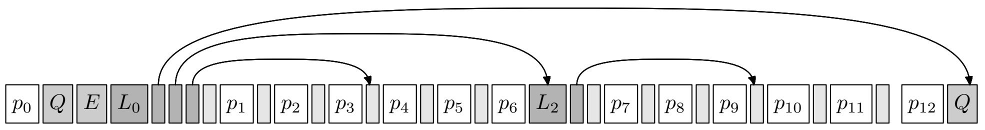

Figure 7: The block shown in Figure 4 with tower top elimination.

The above computation leads us the following, fundamental observation: *a non-truncated tower with highest entry $\bar{h}$ inherits an entry of level $\bar{h}+1$ that is identical to its top entry.* As a consequence, if lists are traversed from their beginning, top entries of non-truncated towers can be omitted. The omission of top entries halves the number of entries written, and, as we observed at the start of the paragraph, reduces even further the skip structure size. The final form of an embedded perfect skip list is shown in Figure 7.

5 Writing Blocks and Towers

It is now time to explain how a block containing a skip structure can be actually written. When writing out an inverted index, data relative to a single block are retained in a cache and written out only at the end of the block, or when the inverted list is over: the limiting parameter h is fundamental in containing the size of this cache, which should be kept in core memory for faster processing (actually, only pointers and block lengths are necessary to compute the skip structure). At this point, we know the average quantum bit length Q, but we do not know the the average number of bits per tower entry E. There is a form of recursion here, because the way skip towers are coded depends on the length in bits that will be necessary to code them. As a matter of fact, the value E does not need to be really the average number of bits per entries, but a value that is close enough will certainly reduce the overall space occupied by skip towers and to make the prediction described in Section 3 more precise.

Given a tentative value $E_0 = 0$ for E, we compute the tower entries from the end: in this way, whenever we have to write a bit skip in a tower entry, we know exactly the number of bits it must specify, and we can also predict it using the formulae described in Section 3. At the end of this process, we compute the number of bits E_1 that a tower entry occupies on the average; then, we try again and compute E_2, etc. This series of attempts stops as soon as the number of bits predicted coincides with the number of bits actually written; in any case, there is an upper bound on the number of possible attempts. At the end, the value of E that is actually used is the one that produced the best compression ratio (in practise, no more than a couple of attempts are actually necessary). Once the value of E is fixed, the block can be written.

6 Experimental data

We gathered statistics indexing a partial snapshot of 13 Mpages taken from the `.uk` domain containing about 50GiB of parsed text (the index contained counts and occurrences). We report some preliminary date gathered using disk-based indices: more investigation is needed with significantly larger collections, and testing both disk-based and in-memory indices.

The document distribution in the snapshot was highly skewed, as documents appeared in crawl order. Adding an embedded perfect skip list structure with arbitrary tall towers caused an increase in size of 2.3% (317 MiB) with $q = 32$ and 1.23% when $q = 64$; indexing one every $\sqrt{f_t}$ elements caused an increase of 0.85%.

Compressing the same skip structures using a γ or δ code instead of Gaussian Golomb codes for pointer skips caused an increase in pointer-skip size of 42% and 18.2%, respectively. This shows the efficiency of Gaussian Golomb codes; however, if the quantum is not very small, due to the small footprint of embedded skip list δ codes might be a viable choice.

Speed is, of course, at the core of our interests. The bookkeeping overhead of skip lists increases by no more than 5% (and by .5% on the average) the time required to perform a linear scan. On the contrary, tests performed on synthetically generated queries in disjunctive normal form show an increase in speed between 20 and 300% w.r.t. the classical (square-root spaced) approach.

7 Conclusions

All in all, compressed embedded perfect skip lists are a simple and elegant way to skip quickly in an inverted list. They have a very small footprint, and nonetheless provide logarithmic-time access to each quantum. Future research will concentrate on obtaining even better codes for highly skewed collection, on a more comprehensive set of statistical tests based on real-world search-engine queries, and on comparisons with advanced indexing systems such as those described in [2].

References

[1] A. Anagnostopoulos, A. Z. Broder, and D. Carmel. Sampling search-engine results. In *Proc. of the Fourteenth International World Wide Web Conference*, Chiba, Japan, 2005. ACM Press.

[2] Vo Ngoc Anh and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In W. Bruce Croft, Alistair Moffat, C. J. van Rijsbergen, Ross Wilkinson, and Justin Zobel, editors, *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR-98)*, pages 290–297, New York City, 1998. ACM Press.

[3] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] Solomon W. Golomb. Sources which maximize the choice of a Huffman coding tree. *Inform. and Control*, 45(3):263–272, 1980.

[5] Neri Merhav, Gadiel Seroussi, and Marcelo J. Weinberger. Optimal prefix codes for sources with two-sided geometric distributions. *IEEE Trans. Inform. Theory*, 46(1):121–135, 2000.

[6] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[7] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In Frances Frederickson, Grag; Graham, Ron; Hochbaum, Dorit S.; Johnson, Ellis; Kosaraju, S. Rao; Luby, Michael; Megiddo, Nimrod; Schieber, Baruch; Vaidya, Pravin; Yao, Andy, editor, *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '92)*, pages 367–375, Orlando, FL, USA, 1992. SIAM.

[8] William Pugh. A skip list cookbook. Technical report UMIACS-TR-89-72.1, Univ. of Maryland Institute for Advanced Computer Studies, College Park, College Park, MD, USA, 1990.

[9] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[10] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[11] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

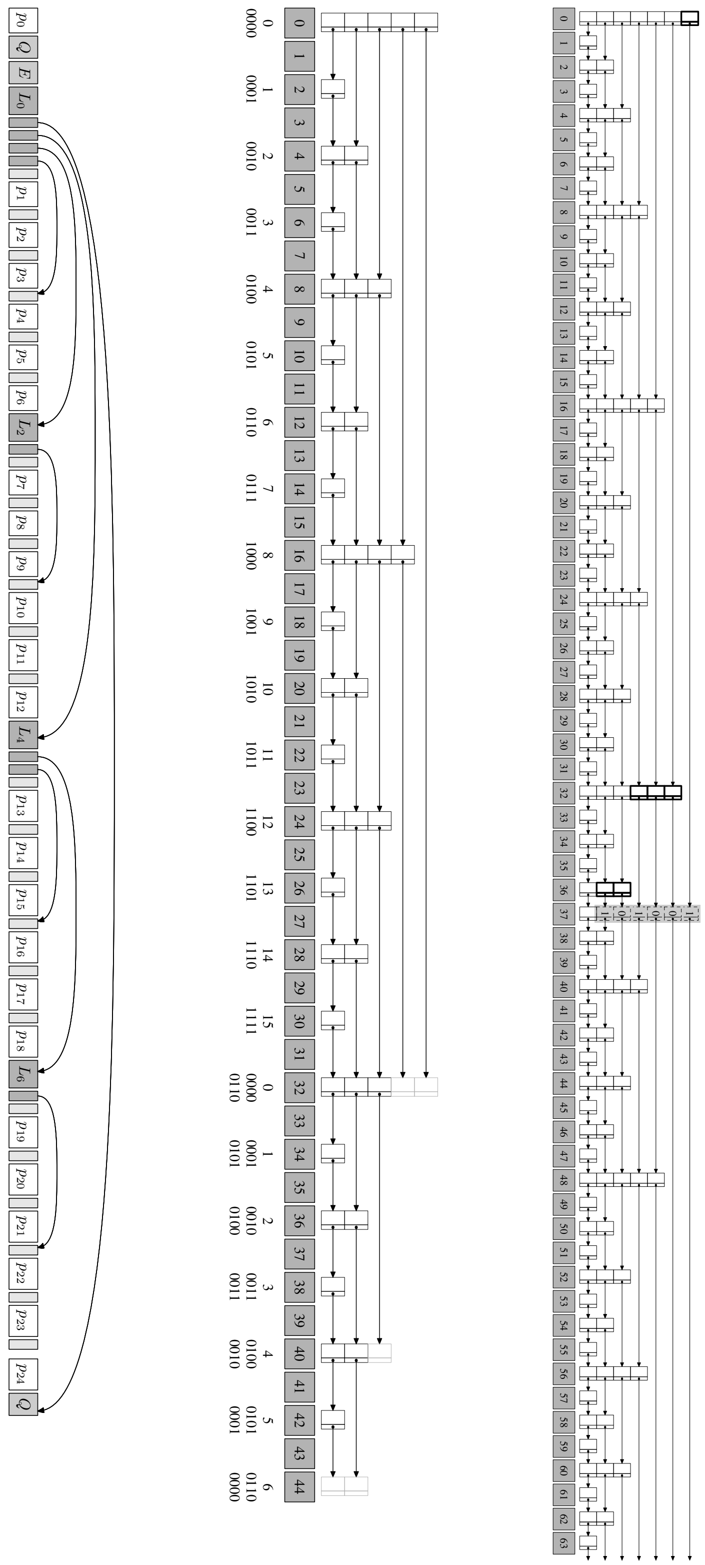

Figure 8: Additional examples of skip lists.