BitFunnel: Revisiting Signatures for Search

Bob Goodwin
Microsoft

Michael Hopcroft
Microsoft

Dan Luu
Microsoft

Alex Clemmer
Heptio

Mihaela Curmei
Microsoft

Sameh Elnikety
Microsoft

Yuxiong He
Microsoft

ABSTRACT

Since the mid-90s there has been a widely-held belief that signature files are inferior to inverted files for text indexing. In recent years the Bing search engine has developed and deployed an index based on bit-sliced signatures. This index, known as BitFunnel, replaced an existing production system based on an inverted index. The driving factor behind the shift away from the inverted index was operational cost savings. This paper describes algorithmic innovations and changes in the cloud computing landscape that led us to reconsider and eventually field a technology that was once considered unusable. The BitFunnel algorithm directly addresses four fundamental limitations in bit-sliced block signatures. At the same time, our mapping of the algorithm onto a cluster offers opportunities to avoid other costs associated with signatures. We show these innovations yield a significant efficiency gain versus classic bit-sliced signatures and then compare BitFunnel with Partitioned Elias-Fano Indexes, MG4J, and Lucene.

CCS CONCEPTS

• **Information systems** → **Search engine indexing**; *Probabilistic retrieval models*; *Distributed retrieval*; • **Theory of computation** → Bloom filters and hashing;

KEYWORDS

Signature Files; Search Engines; Inverted Indexes; Intersection; Bitvector; Bloom Filters; Bit-Sliced Signatures; Query Processing

1 INTRODUCTION

Commercial search engines [2, 5, 19, 24] traditionally employ inverted indexes. In this work, we show how to use signatures, or bit-strings based on Bloom filters [1], in a large-scale commercial search engine for better performance. Prior work comparing inverted files to signature files established that inverted files outperformed signature files by almost every criterion [28]. However, recent software and hardware trends (e.g., large Web corpora with of billions of documents, large main memory systems) motivated us to reconsider signature files.

In our signature-based approach, known as BitFunnel, we use a Bloom filter to represent the set of terms in each document as a fixed sequence of bits called a *signature*. Bloom filters are reasonably space efficient and allow for fast set membership, forming the basis for query processing.

Using this approach, however, poses four major challenges. First, determining the matches for a single term requires examining one signature for each document in the corpus. This involves considerably more CPU and memory cycles than the equivalent operation on an inverted index. Second, term frequency follows a Zipfian distribution, implying that signatures must be long to yield an acceptable false positive rate when searching for the rarest terms. Third, the size of web documents varies substantially, implying that signatures must be long to accommodate the longest documents. Fourth, the configuration of signature-based schemes is not a well-understood problem.

We develop a set of techniques to address these challenges: (1) we introduce *higher rank rows* to reduce query execution time; (2) we employ *frequency-conscious signatures* to reduce the memory footprint; (3) we shard the corpus to reduce the variability in document size; (4) we develop a cost model for system performance; and (5) we use this model to formulate a constrained optimization to configure the system for efficiency.

These techniques are used in the Microsoft Bing search engine, which has been running in production for the last four years on thousands of servers. Compared to an earlier production search engine based on inverted lists that it replaced, BitFunnel improved server query capacity by a factor of 10.

2 BACKGROUND AND PRIOR WORK

We focus on the problem of identifying those documents in a corpus that match a conjunctive query of keywords. We call this the *Matching Problem*.

Let corpus $\mathbb{C}$ be a set of documents, each of which consists of a set of text terms:

$$\mathbb{C} = \{\text{documents } D\}$$
$$D = \{\text{terms } t\}$$

Define query Q as a set of text terms:

$$Q = \{\text{terms } t\}$$

SIGIR '17, August 07-11, 2017, Shinjuku, Tokyo, Japan

ACM ISBN 978-1-4503-5022-8/17/08…$15.00
https://doi.org/10.1145/3077136.3080789

Query Q is said to match document D when every term $t \in Q$ is also an element of D. This happens when $Q \subseteq D$ or $Q = Q \cap D$. Define match set M as the set of documents matching Q:

$$M = \{D \in \mathbb{C} \mid Q = D \cap Q\}$$

The goal of the Matching Problem is to identify the match set M, given corpus $\mathbb{C}$ and query Q.

In Sections 2.2-2.4 we examine conservative probabilistic algorithms that never miss a match, but might falsely report matches. The goal for these algorithms is to identify a conservative filter set M'

$$M \subseteq M' \subseteq \mathbb{C}$$

where the false positive set $F = M' \setminus M$ is small.

2.1 Inverted Indexes

Perhaps the most common approach to the *Matching Problem* is the *inverted index* [4, 11]. This approach maintains a mapping from each term in the lexicon to the the set of documents containing the term. In other words,

$$Postings(t) = \{D \in \mathbb{C} \mid t \in D\}$$

With this approach, M can be formed by intersecting the posting sets associated with the terms in the query:

$$M = \bigcap_{t \in Q} Postings(t)$$

In practice, the posting sets are usually sorted, allowing fast intersection. They also draw on a large bag of tricks [4, 20] to compress and decompress posting sets [17, 23] while improving intersection time [6, 7]. This is a rich area with ongoing research into novel data structures such as treaps [16] and semi-bitvectors [13].

Inverted indexes find the exact match set, M, every time. Signature-based approaches [8–10, 15, 25], on the other hand, use probabilistic algorithms, based on superimposed coding [1, 21, 22] and newer approaches, like TopSig [12] to identify a conservative filter set M'. BitFunnel is based on classical bit-sliced signatures which are, in turn, based on bit-string signatures.

2.2 Bit-String Signatures

The key idea is that each document in the corpus is represented by its *signature*. In BitFunnel, the signature is essentially the sequence of bits that make up a Bloom filter representing the set of terms in the document. In constructing the Bloom filter, each term in the document is hashed to a few bit positions, each of which is set to 1.

Let n denote the number of bit positions in the Bloom filter. Define $H(n, t)$ as a function that returns the set of bit positions in the range $[0..n)$ corresponding to the hashes of term t. Define $\vec{s_t}$, the signature of term t, as the bit-vector of length-n where bit position i is set to 1 iff $i \in H(n, t)$. We can then define the signature of document D as the logical-or of the signatures of its terms:

$$\vec{s_D} = \bigcup_{t \in D} \vec{s_t}$$

In a similar manner, we can define the signature of query Q as the logical-or of the signatures of its terms:

$$\vec{s_Q} = \bigcup_{t \in Q} \vec{s_t}$$

Document D is said to be a member of M' when

$$\vec{s_Q} \cap \vec{s_D} = \vec{s_Q}$$

Given the signatures of the documents in the corpus, one can easily compute M' by identifying those documents whose signatures match the query's signature:

$$M' = \{D \in \mathbb{C} \mid \vec{s_Q} \cap \vec{s_D} = \vec{s_Q}\}$$

Here's the pseudocode to search a corpus for documents matching a query:

```
M' = ∅
for all D ∈ ℂ do
    if s_D ∩ s_Q = s_Q then
        M' = M' ∪ {D}
    end if
end for
```

Bit-string signatures are elegant, but their uniform encoding of terms, independent of frequency, leads to poor memory utilization. Section 4.2 explains how BitFunnel uses *Frequency Conscious Signatures* to improve memory efficiency in signatures.

2.3 Bit-Sliced Signatures

If all of the signatures have the same length and share a common hashing scheme, $H(n, t)$, one can achieve significant performance gains by using a bit-sliced arrangement [9, 26, 27]. This approach transposes signature vectors from rows to columns in order to allow multiple documents to be searched simultaneously while eliminating the bit masks and shifting necessary to perform Boolean operations on individual bits.

Suppose we have a corpus $C = \{A..P\}$ and a query Q. The matrix in Figure 1 shows these documents and the query encoded as bit-sliced signatures. Each document corresponds to a column which holds its 16-bit signature. Each row corresponds to a bit position in the document signature.

In this example the signature for document B has bit positions 2, 5, 9, and 13 set. The signature for the query Q has bit positions 2, 5, and 9 set. Therefore, document B will match the query. It turns out that document F also matches the query.

With the bit-sliced layout, we need only inspect the rows corresponding to bit positions in Q that are set. These rows, which we call the query's rows, are shaded in Figure 1 and isolated in Figure 2. Each bit position in the query's rows corresponds to a document. The document matches if its bit position is set in all of the query's rows. We determine which documents match by intersecting the query's rows and looking for set bits. In Figure 2, columns B and F are the only columns without zeros. Therefore documents B and F are the only matches.

Here's the bit-sliced algorithm:

```
a = ~0
for all i where s_Q[i] == 1 do
    a = a & row_i
end for
M' = {i | a[i] ≠ 0}
```

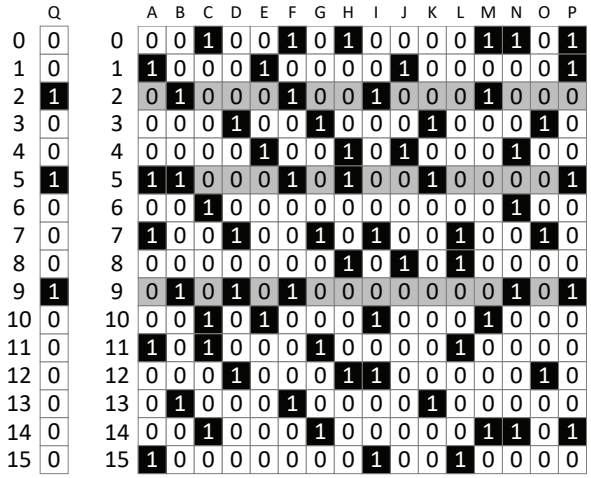

Figure 1: Layout with bit-sliced signatures, in which each column is a document signature. Q is the signature of the query.

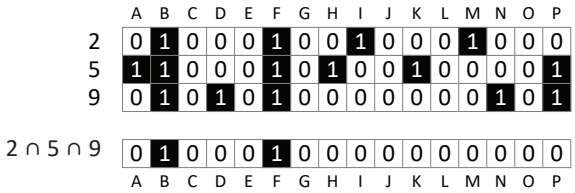

Figure 2: Bit-sliced signature layout. Rows 2, 5, and 9 yield documents B and F.

2.4 Bit-Sliced Blocked Signatures

While bit-sliced signatures offer a significant performance advantage over bit-string signatures, they still suffer from poor performance when searching for rare terms. The problem is that every document's bit position must be scanned, even in the case where only a handful of documents actually match.

The idea behind blocked signatures [14] is to create shorter rows by assigning multiple documents to each column in the signature matrix. The number of documents that share a column is called the *blocking factor*. Shorter rows improve performance because they can be scanned more quickly, but they introduce noise which increases the false positive rate.

Prior to BitFunnel, bit-sliced block signatures were used primarily as a single-level index into a set of bit-string signature files on disk. At the time the main concern with this approach was reducing the probability of an *unsuccessful block match* which occurred when a column signature matched the query but none of the documents contained all the terms in the query. Suppose, for example, a column held two documents, one containing the word "dog" and the other containing the word "cat". This column would match the query {"*dog*", "*cat*"} even though neither document contains both terms. At least one paper proposed a solution to the problem of unsuccessful block matches [14], however [28] argued that blocking increases complexity while offering little benefit. In Section 4.1, we introduce *Higher Rank Rows* to address these problems.

3 THE BITFUNNEL SYSTEM

For the past 4 years, BitFunnel has powered Bing's fresh index of recently crawled documents. During this time the system, which runs on thousands of machines, spread across several data centers, has processed the entire query load sent to Bing.

3.1 Architectural Overview

Bing maintains multiple, georeplicated copies of the web index, each of which is sharded across a cluster of BitFunnel nodes. Figure 3 shows a single cluster. Queries are distributed, round robin, across the cluster. A node, upon receiving a query, parses it into an abstract syntax tree, rewrites the tree into an execution plan and then compiles the plan locally before broadcasting the compiled plan to the rest of the cluster. The nodes in the cluster run the compiled plan in parallel, returning results to the planning node for aggregation. These results are then passed on to other systems that score the matching documents and generate captions to display on the search results web page.

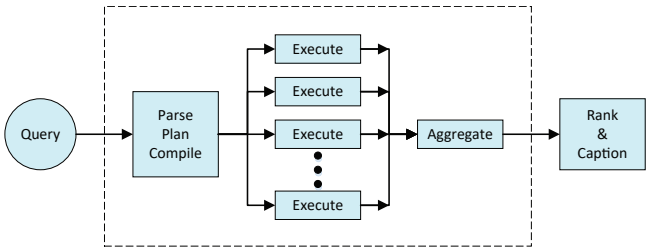

Figure 3: BitFunnel cluster.

3.2 The Cost of False Positives

One criticism specific to the signature file approach is the introduction of false positives into the result set. For scenarios like database queries where the identifying exact match set is the goal, the cost of filtering out the false positives can be prohibitive. In the case of web search, the cost of filtering out false positives is negligible. To see why, it is important to understand that the goal of web search is not to find documents matching Boolean expressions of keywords – rather it is to find the documents that best match the user's intent when issuing a query. In Bing, we employ a ranking system that, given a document and a query, will generate a score predicting how well the document matches the user's intent for the query. This system relies on many signals beyond keywords and to some extent its inner workings are opaque to us because it is configured by machine learning.

If we had unlimited resources, we could process each query by submitting every single document in the corpus to our ranking oracle and then return the top-n ranked documents. Since we don't have unlimited resources, we insert inexpensive filters upstream of the oracle to discard documents that the oracle would score low. The filters are designed to reject, with high probability, those

documents that score low while never rejecting documents that score high. BitFunnel is such a filter.

In this context, the performance of BitFunnel is judged by its impact on the end-to-end system. BitFunnel wins when its time savings in the Boolean matching phase is greater than the time the oracle spends scoring false positives.

We turn our attention now to a single BitFunnel node to describe the techniques that enable fast query processing.

4 BITFUNNEL INNOVATIONS

In this section, we describe three innovations that address speed and space problems associated with bit-string and bit-sliced signatures.

4.1 Higher Rank Rows

BitFunnel generalizes the idea of blocking so that each term simultaneously hashes to multiple bit-sliced signatures with different blocking factors. The key to making this approach work is the ability to efficiently intersect rows with different blocking factors.

4.1.1 Mapping Columns Across Ranks. In BitFunnel, we restrict blocking factors to be powers of 2. We define a concept of row *rank*, where a row of rank $r \geq 0$ has a blocking factor of 2^r.

The BitFunnel blocking scheme is specific to the size of the machine word used for the bit-slice operations. Let w be the $\log_2$ of the number of bits in a machine word, so for example, a 64-bit processor would have $w = 6$. Then the document in column i_0 at rank 0 will be associated with column i_r at rank r as follows:

$$i_r = \frac{i_0}{2^{r+w}} + (i_0 \bmod 2^r) \tag{1}$$

Figure 4 gives a small example for a 4-bit machine word ($w = 2$) and ranks 0, 1, and 2. We can see that position 4 at rank 1 is associated with documents $\{4, 12\}$ while position 0 at rank 2 is associated with documents $\{0, 4, 8, 12\}$.

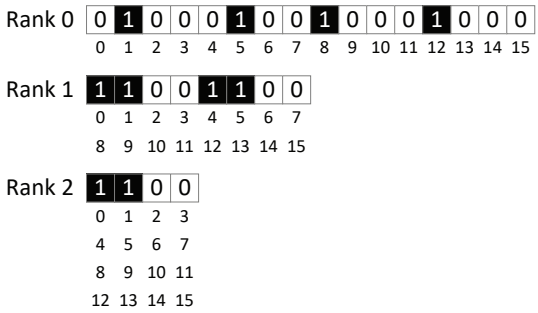

Figure 4: Forming higher rank equivalents of a single row.

Note that higher rank rows will, in general, magnify the bit density of their lower rank equivalents. This is because the value of each bit at a higher rank is the logical-or of multiple bits at a lower rank. In order to maintain a constant bit density of d across all signatures in BitFunnel, we must use longer signatures at higher ranks. Therefore, a single row at rank 0 will translate into multiple shorter rows at a higher rank. In most cases, a rank zero row and its higher rank equivalents will consume roughly the same amount of memory. We will derive an expression for the memory consumption in higher rank rows in Section 5.4.

Now suppose we have a query, Q, that maps to the three rows shown in Figure 5. To evaluate the query, we need some way of intersecting rows with different ranks. The mapping in Equation (1) is designed to make this operation easy and efficient.

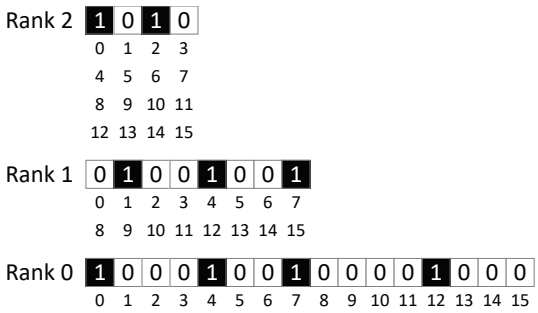

Figure 5: Intersecting different rows with different ranks.

Logically we convert each row to its *rank-0 equivalent* by concatenating 2^r copies of the row as shown in Figure 6. Then we are free to intersect the rank-0 equivalent rows to produce the result vector.

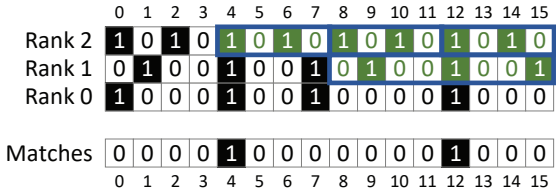

Figure 6: Rank-0 equivalent rows.

4.1.2 Optimizing Higher Rank Row Intersection. At a logical level, our approach is to intersect rank-0 equivalent rows. Were we to generate rank-0 equivalents for intersection, we would lose all of the performance gains that come from scanning shorter rows at higher ranks. Mapping (1) was structured specifically to provide opportunities to reuse intermediate results at every rank. As an example, in Figure 6, bits [0..3] of the rank 2 row need only be read once, even though they will be used for positions [4..7], [8..11], and [12..15]. Similarly, the intersection of the first two rows in positions [0..3] will be computed once and used again for positions [8..11]. We will leverage this insight in Section 5.3 where we derive an expression for the expected number of operations required to combine a set of rows with different ranks.

In BitFunnel, each term in a query maps to a set of rows that may include higher rank rows.

4.2 Frequency Conscious Signatures

We saw in Section 2.2 how Bloom filter signatures can be used to encode the set of terms in a document. One shortcoming with this approach is inefficient memory usage when terms in the lexicon have widely varying frequencies in the corpus.

The problem stems from the fact that, in its classical formulation [1], the Bloom filter is configured with an integer constant, k, which represents the number of hashes for each term[1]. This value of k is the same for all terms in lexicon L. In other words

$$|H(n,t)| = k, \ \forall_{t \in L}$$

To get an intuition for the problem of the one-size-fits-all k, it helps to think of the quantity of false positives in terms of signal-to-noise ratio. Let's consider a single set membership test for $t \in D$. In the context of the test, define the signal s to be the probability that term t is actually a member of document D. This is just the frequency of t in the corpus.

Define noise α to be the probability that the Bloom filter will incorrectly report t as a member. Assume the Bloom filter has been configured to have an average bit density of d. Since d is the fraction of the bits expected to be set, we can treat it as the probability that a random bit is set. A set membership test involves probing k bit positions. If all k probes find bits that are set to one, the algorithm will report a match. Therefore the noise is just the probability that k probes all hit ones when $t \notin D$:

$$\alpha = (1-s)d^k$$

The signal-to-noise ratio ϕ is then

$$\phi = \frac{s}{(1-s)d^k}$$

We can rearrange this and take the ceiling to get an expression for k as a function of d, s, and ϕ:

$$k = \left\lceil log_d\left(\frac{s}{(1-s)\phi}\right)\right\rceil$$

This is the minimum value of k that will ensure a signal-to-noise ratio of at least ϕ. The main take away is that k increases as s decreases. In other words, rare terms require more hashes to ensure a given signal-to-noise level. The following table shows values of k without the ceiling, for select values of s when $d = 0.1$ and $\phi = 10$:

signal (s)	hashes (k)
0.1	1.954242509
0.01	2.995635195
0.001	3.999565488
0.0001	4.999956568
0.00001	5.999995657

Now consider a Bloom filter that stores a typical document from the Gov2 corpus[2]. If we were to configure the Bloom filter with $k = 2$ we could just barely maintain a signal-to-noise ratio of 10 when testing for the term "picture" which appears with frequency 0.1. To test for the term "rotisserie", which appears with frequency 0.0001, we would need $k = 5$ to drive the noise down to a tenth of the signal.

With classical Bloom filters, one must configure for the rarest term in the lexicon, even though the vast majority of common terms could be stored more efficiently. Recent work in Weighted Bloom Filters [3] shows that it is possible to adjust the number of hash functions on a term-by-term basis within the same Bloom filter.

[1]In Bloom's original paper [1] this constant was the letter d; more contemporary descriptions [3] use the letter k.
[2]Term frequencies are from Corpus D described in Section 6.

BitFunnel applies these ideas to reduce memory usage and determine the number of rows needed for each term.

4.3 Sharding by Document Length

Bit-sliced signatures have another one-size-fits-all problem resulting from the requirement that all of the document signatures have the same configuration (i.e. their bit lengths, n, must all be the same, and they must all use the same hashing scheme $H(n,t)$).

The problem is that real world documents vary greatly in length. In Wikipedia, for example, the shortest documents have just a handful of unique terms while the longest ones may have many thousands of terms. The dynamic range of document lengths on the internet is even higher because of files containing DNA sequences, phone numbers, and GUIDs. To avoid overfilling our Bloom filters and generating excessive false positives, it is necessary to configure the Bloom filters for the longest document expected, even if such a document is very rare. Unfortunately, such a configuration would waste enough memory as to offset all of the other benefits of the bit-sliced arrangement.

A workaround [28] suggested in the late 90s was to shard the index into pieces containing documents with similar lengths. This approach was rejected at the time because, on a single machine, the introduction of length sharding would multiply the number of disk seeks by the number of shards.

This concern is not a factor when the index is many times larger than the capacity of a single machine. As soon as the index is sharded across a large cluster, one must pay for the overhead of sharding. At this point sharding by document length costs the same as sharding by any arbitrary factor.

Even on a single machine, the cost of length sharding is greatly reduced on modern hardware where the index can be stored in RAM or on SSD because the access cost is dominated by fixed-sized block transfers (512-bit cache line for RAM, 4096 byte block for SSD), rather than hard disk seeks.

In BitFunnel, we partition the corpus according to the number of unique terms in each document such that each instance of BitFunnel manages a shard in which documents have similar sizes.

5 PERFORMANCE MODEL AND OPTIMIZATION

Signature-based approaches have historically been hard to configure because of a large number of design parameters that impact performance [10, 26, 28]. In this section we present an algorithm that optimizes the BitFunnel configuration, given a desired signal-to-noise ratio. The algorithm performs a constrained optimization, over relevant configuration parameters, of a cost function that expresses the system efficiency as DQ, the product of the corpus size D and query processing rate Q. The configuration parameters include the mapping from terms with various frequencies to their corresponding number of rows at each rank. The constraint is a lower limit on the allowable signal-to-noise ratio, ϕ.

In order to develop the cost function and constraint, we derive expressions for the signal-to-noise ratio, query processing speed, and memory consumption in BitFunnel. We then combine these expressions into a cost function and constraint used by the algorithm that identifies an optimized set of blocking factors and hash

functions for each equivalence class of terms, based on frequency in the lexicon.

5.1 Prerequisites

Before deriving these fundamental equations, we discuss the impact of row rank on bit densities and noise. We then characterize two different components of noise in rank-0 equivalent rows. This will form the basis for the noise, speed, and storage equations in Sections 5.2, 5.3, and 5.4.

5.1.1 Signal in a Higher Rank Row. Because each bit in a higher rank row corresponds to multiple documents, the bit density contributed by a single term will nearly always be greater in higher rank rows. We can see this in Figure 4 where densities in the rank-0 row and its rank 1 equivalent are $\frac{4}{16}$ and $\frac{8}{16}$, respectively.

Let s_0 denote the signal in a rank-0 row and s_r denote the signal at rank r. We can express s_r as a function of s_0 and r. The probability that a bit at rank r is set due to signal is the probability that at least one of the 2^r corresponding rank-0 bits is signal. This is just one minus the probability that all of the 2^r rank-0 bits are zero:

$$s_r = 1 - (1 - s_0)^{2^r} \tag{2}$$

5.1.2 Noise in a Rank-0 Equivalent Row. Processing a query in BitFunnel is logically equivalent to intersecting the rank-0 equivalents of each row associated with the query. Converting a rank-r row to its rank-0 equivalent increases noise. The intuition behind this is simple — each bit set in a rank-r row means that at least one of 2^r documents is a match. It could be one document or all 2^r — we can't tell and this is the source of higher noise.

Let's look at a simple example. Suppose we have a corpus of 16 documents and would like to search for a term that happens to appear in documents 4 and 8. We hash our term to find its corresponding rows, and we get the set of rows $R = \{R_2, R_1, R_0\}$ with ranks 2, 1, and 0, respectively. We define the signal, s_0 as the fraction of the bit positions at rank-0 corresponding to a match. In the case of a term that appears in only 2 documents, $s_0 = \frac{2}{16}$. In Figure 7, the green squares labeled 'S' correspond to the signal.

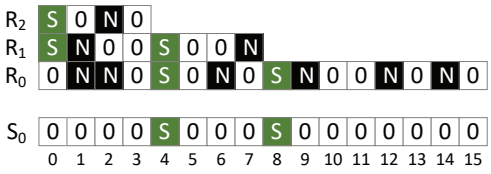

Figure 7: A term maps to three rows with different ranks. Since a row is shared with other terms, it contains signal and noise bits but has constant bit density.

R_2 has one signal bit, so its signal is $\frac{1}{4} = \frac{4}{16}$. We've arbitrarily added one noise bit, marked with an 'N' and shaded black. This bit is contributed from another term that also maps to R_2. The density of R_2 is $\frac{2}{4} = \frac{8}{16}$.

Row R_1 has two signal bits and two arbitrary noise bits so its signal is $\frac{2}{8} = \frac{4}{16}$ and its density is $\frac{4}{8} = \frac{8}{16}$.

Finally, in row R_0, the signal is equal to s_0 because each signal bit maps directly to a single document. As with the other rows, R_0 contains random noise bits from other terms, yielding $\frac{8}{16}$ density.

To process our query, we intersect the rank-0 equivalents of rows R_2 and R_1 with R_0. Figure 8 shows how the process of creating rank-0 equivalents increases noise.

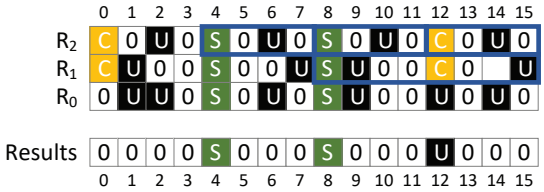

Figure 8: Noise in rank-0 equivalent rows.

Continuing with our example, the signal bit from position 0 in R_2 maps to bit positions 0, 4, 8, and 12 at rank 0. Of these four positions, only positions 4 and 8 correspond to signal bits. The others are noise bits introduced by the construction of the rank-0 equivalent, and they are colored yellow and marked with the letter 'C'. In a similar manner, R_2 bit position 2 introduces noise in rank-0 positions 2, 6, 10, and 14. These bits are colored black and marked with the letter 'U'. In the case of R_2, we went from a rank-2 row with $\frac{1}{4}$ signal and $\frac{1}{4}$ noise to a rank-0 row with $\frac{2}{16}$ signal and $\frac{6}{16}$ noise. The noise increase is entirely due to the signal bits in R_2. In contrast, the noise bits in R_2 contribute their same density without amplification, and therefore do not increase noise in the rank-0 equivalent row.

Now let's look at the rank-0 equivalent of R_1. We go from a rank-1 row with $\frac{2}{8}$ signal and $\frac{2}{8}$ noise to a rank-0 row with $\frac{2}{16}$ signal and $\frac{6}{16}$ noise. As with R_2, the noise increase is due entirely to signal in rank-1 row.

5.1.3 Correlated and Uncorrelated Noise. We turn to computing the noise resulting from the intersection of a set of the rows. The noise in any rank-0 row is the difference between the row's density and the signal s_0. If row R has density d, then its rank-0 equivalent has density d because it consists of the concatenation of 2^r copies of the R. Therefore, the noise in R's rank-0 equivalent is $d - s_0$.

Noise is made up of two components, one which is correlated and one which is not. In Figure 8, uncorrelated noise bits are shaded black while correlated noise bits are colored yellow. Row intersections are very effective at reducing uncorrelated noise, but they have less impact on correlated noise.

To better illustrate this, let's look at a simple, but extreme example. Suppose our query matches documents 2 and 13 and consists of the three rank-1 rows depicted in Figure 9. In the rank-0 equivalent

Figure 9: Three rank-2 rows.

rows, shown in Figure 10, noise has two components: correlated and

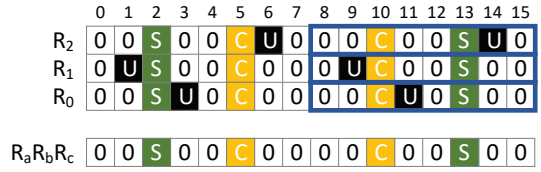

Figure 10: Correlated and uncorrelated noise in rank-0 equivalent rows.

uncorrelated. The uncorrelated noise, shown in black and marked with the letter 'U', is completely eliminated in three row intersections, but the correlated noise, shown in yellow and marked with the letter 'C' remains at the same level despite the intersections.

Effectively managing the impact of higher rank rows requires an understanding of the correlated noise in rank-0 equivalent rows. In the following we derive expressions for noise components.

Let n_r denote noise in a rank-r row R and n_0 denote noise in its rank-0 equivalent. We express n_0 as a function of r, s_0, and n_r. Row R will contribute n_r density due to noise already in R and s_r in density due to signal in R. A portion of the density in s_r corresponds to bonified signal. The remaining density is correlated noise introduced by the conversion to rank-0. Thus we compute noise at rank-0 by subtracting s_0 from the density contributed by R:

$$n_0 = n_r + s_r - s_0$$

To compute the correlated noise, we subtract n_r from n_0 and substitute $s_r = 1 - (1 - s_0)^{2^r}$:

$$n_0 - n_r = s_r - s_0 = 1 - (1 - s_0)^{2^r} - s_0 \tag{3}$$

Note that the number of correlated noise bits in a rank-0 equivalent is a function of the original rank. The higher the row rank, the greather the contribution in correlated noise to its rank-0 equivalent. Also, correlated noise remaining after intersecting a set of rank-0 equivalents is the correlated noise of the lowest rank row in the set. The other correlated noise is converted to uncorrelated noise.

It is important to note that the correlated noise bits in a lower rank equivalent always form a subset of the correlated noise bits in a higher rank equivalent. Our equations for noise and speed in Sections 5.2 and 5.3 make use of this fact.

5.2 Signal-to-Noise Ratio

We're now ready to write expressions for the noise components after a sequence of row intersections. For this derivation, we will perform the intersections in order from high rank to low rank. We will start with an accumulator, a, which has an initial bit density of 1.0 and then intersect in each row in turn.

Let a_i denote the total noise in the accumulator at the end of iteration i. Let c_i and u_i denote the amount of correlated and uncorrelated noise, respectively, on iteration i and let r_i denote the rank. The first iteration is effectively loading the first row into the accumulator so, $u_1 = n_1$. The correlated noise in the accumulator is always equal to the correlated noise in the last row intersected, so

$$c_i = 1 - (1 - s_0)^{2^{r_i}} - s_0$$

Since the rows are ordered by non-increasing rank, subsequent rows will never have more correlated noise. In the case where the rank decreases, the amount of correlated noise will decrease. When this happens, some of the correlated noise in the accumulator will become uncorrelated noise, moving forward. This new amount of uncorrelated noise in the accumulator will then be multiplied by the current row's total noise density n_{i+1}:

$$u_{i+1} = (u_i + c_i - c_{i+1})n_{i+1}$$

At any given point, the total accumulator noise a_i is just the sum of the correlated and uncorrelated noise:

$$a_i = c_i + u_i$$

The signal-to-noise ratio, ϕ, on iteration i is then

$$\phi_i = \frac{s_0}{a_i} = \frac{s_0}{c_i + u_i} \tag{4}$$

5.3 Query Execution Time

When modelling running time, we use the number of machine word accesses of unique memory addresses as our proxy for time. On a real computer, row intersections are typically performed in chunks that match the machine register size. As an example, if the machine register size is 64 bits, and the rank-0 rows are 256 bits long, a pairwise row intersection would require 4 register-sized logical-and operations. When intersecting a set of rows, the outer loop is typically over the register-sized chunks in each row and the inner loop is over the set of rows.

This ordering of the loops is desirable because intermediate results of row intersections can reside in the accumulator instead of being written to memory. In many cases, the accumulator will become zero in the inner loop before all of the rows have been examined. Since additional intersections cannot change the result, it is possible to break out of the inner loop at this point.

In practice, breaking out of the inner loop offers a significant performance improvement. To quantify this impact, we'll focus on the innermost loop which intersects a set of n machine words that reside in memory. Our goal is to write an expression for the expected number of machine words loaded from memory.

If we know the probability that a bit remains set after intersecting the first n rows, we can derive a formula for the expected number of machine words accessed when intersecting a set of rows.

Let N be a random variable denoting the machine words intersected and define $P_{BZ}(N > i)$ to be the probability that a random bit in the accumulator is zero after iteration i. $P_{BZ}(N > i)$ is the probability that the bit was not set by noise and not set by signal:

$$P_{BZ}(N > i) = 1 - s_0 - a_i$$

Define $P_A(N > i)$ to be the probability that at least one bit in the accumulator remains set after i intersections If b denotes the number of bits in a machine word then

$$P_A(N > i) = 1 - (P_{BZ}(N > i))^b$$

If we were to actually perform intersections on the rank-0 equivalent rows, the expected number of machine words accessed during one iteration of the outer loop would be

$$\mathbb{E}(N) = \sum_{i=1}^{n} P_A(N > i) = \sum_{i=1}^{n} 1 - (1 - s_0 - a_i)^b$$

As we saw in Section 4.1.2, the mapping of columns across ranks is structured in such a way that intermediate results from higher rank intersections can be reused. Since each rank-0 equivalent is just the concatenation of 2^r copies of a rank-r original, we need only load the accumulator once for each of the 2^r machine word positions in the rank-0 equivalent. This reduces the number of machine words accessed in each row by a factor of 2^{r_i}:

$$\mathbb{E}(N) = \sum_{i=1}^{n} \frac{1 - (1 - s_0 - a_i)^b}{2^{r_i}} \tag{5}$$

A similar approach can be used to model block devices like CPU cache and SSD block transfers, but it is somewhat more involved than substituting a different value for b.

5.4 Space Consumption

We express memory consumption as the number of bits per document required to store a term. Suppose we have a corpus, $\mathbb{C}$, with target bit density, d, and we wish to store a term with signal, s_0, in some row, q, that has rank r.

Since the corpus has $|\mathbb{C}|$ documents, row q must have $|\mathbb{C}|2^{-r}$ bit positions. Equation (2) shows that a term with frequency s_0 will set s_r of these bits. Therefore the term contributes $b_1 = s_r|\mathbb{C}|2^{-r}$ set bits to row q. Let b_0 denote the number of zero bits in row q. By definition,

$$d = \frac{b_1}{b_1 + b_0}$$

Rearranging, we get

$$b_0 = \frac{b_1}{d} - b_1$$

Therefore, the total number of bits required in row q to maintain density of d with a signal of s_0 is

$$b_0 + b_1 = \frac{b_1}{d} = \frac{s_r|\mathbb{C}|}{d2^r}$$

Dividing by the corpus size $|\mathbb{C}|$ gives the number of bits per document signature:

$$\frac{s_r}{d2^r}$$

For a set of rows, Q, the total memory consumption per document is therefore

$$\sum_{q \in Q} \frac{s_r(q)}{d2^r} \tag{6}$$

5.5 Choosing Term Configurations

Given expressions for signal-to-noise ratio, machine word reads, and storage consumed, we can now develop an approach for identifying the optimal row configuration for each term. The problem is a constrained optimization over a cost function parameterized by speed and space. Our constraint is that the signal-to-noise ratio, ϕ, must exceed some fixed threshold. The cost function is proportional to DQ, the product of the number of documents per unit storage and the number of queries processed per unit of compute.

D is inversely proportional to the amount of storage required per document. Q is inversely proportional to the number of machine words accessed while processing a query. Therefore

$$DQ \propto \frac{1}{\left(\sum_{i=1}^{n} \frac{1-(1-s_0-a_i)^b}{2^{r_i}}\right)\left(\sum_{q \in Q} \frac{s_r(q)}{d}\right)} \tag{7}$$

Given the small number of possible row configurations, it is easy to enumerate all configurations and choose the one with the highest DQ where ϕ exceeds the signal-to-noise threshold. For example, when considering configurations of 0 to 9 rows at each of seven ranks from 0 to 6, we need to examine 10^7 configurations for each s_0 value. If we group s_0 values into, say, 100 buckets correspondiong to IDF values from 0.1 to 10.0 in 0.1 increments, the entire optimization involves 10^9 evaluations of Equation 7. A modern multi-core processor can perform this optimization in a matter of seconds.

6 EXPERIMENATAL EVALUATION

Our experiments are based on the TREC Gov2 corpus. Apache Tikka[3] was used to extract terms, which were then converted to lower case, but not stemmed. Since BitFunnel shards its index by document term count, we selected five representative shards for our tests. Shard A has relatively short documents with term counts ranging from 64 to 127. Shards B, C, D and E have progressively larger documents.

Table 1: Corpora.

	A	B	C	D	E
Min terms	64	128	256	1,024	2,048
Max terms	127	255	511	2,047	4,095
Documents (M)	5.870	7.545	3.726	0.494	0.157
Total terms (M)	4.181	6.524	6.647	10.109	9.697
Postings (M)	563	1,411	1,268	687	432
Matches/query	1,115	3,561	5,124	3,728	3,688
Input text (GB)	6.85	25.48	21.02	22.89	20.26

Our query log is based the TREC 2006 Efficiency Topics. We removed punctuations from each query and then filtered out those queries that contained terms not in the corpus.[4] The resulting query log contains about 98k queries.

BitFunnel was implemented in C++14 and compiled with GCC 5.4.1 with the highest optimization level. Experiments were performed on a 4.0GHz 4-core i7-6700 with 32GB of 3.2GHz DDR4 RAM with Ubuntu 14.04 LTS on Windows Subsystem for Linux. BitFunnel was configured with lower bound signal-to-noise ratio $\phi = 10$.

The source code to replicate our experiments is available at http://bitfunnel.org/sigir2017.

6.1 Match Time vs. Quadwords

In Section 5.3 we developed a model for the number of machine words of row data accessed while processing a query. To verify that our model has predictive power, we examined the relationship between row intersection time and the number of quadwords accessed. Since BitFunnel has a significant per-match overhead that

[3] https://tika.apache.org/

[4] This filtering was necessary because the Partitioned Elias-Fano index we used requires all query terms be in the index.

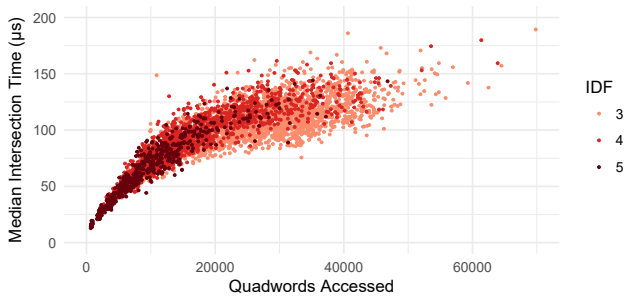

Figure 11: Intersection time increases with quadwords.

is not part of the row intersection cost model, we modified the code to perform row intersections, but not report matches. A sample of 5000 queries with $IDF > 3$ were chosen, at random, from our TREC query log, and these queries were run, single-threaded, against corpus D. To control for system variances not in the model, we ran each query 10 times and recorded the median row intersection time. The scatterplot in Figure 11 shows that row intersection time tends to grow as the number of quadwords increases. The correlation is more pronounced at higher IDF values.

6.2 Impact of Frequency Conscious Signatures and Higher Rank Rows

This experiment compares the time and space characteristics of (a) bit-sliced signatures configured with classical Bloom filters (BSS); (b) the same, but with Frequency Conscious Signatures as described in Section 4.2 (BSS-FC); and (c) Higher Ranked Rows as described in Section 4.1 and optimized per Section 5.5 (BTFNL).[5]

Table 2 examines Corpus D, comparing the three configurations at each of 5 bit densities. The DQ values measure overall system efficiency, expressed as the ratio of QPS to Bits/Posting. We use DQ because it is inversely proportional to the number of servers required, given a particular corpus and a desired QPS

Table 2: Impact of BitFunnel Innovations.

Treatment	Density	Bits/Posting	kQPS	DQ
BSS	0.05	80.0	14.0	175
BSS	0.10	50.0	11.3	225
BSS	0.15	46.7	9.1	194
BSS	0.20	40.0	8.2	204
BSS	0.25	36.0	6.9	191
BSS-FC	0.05	23.4	29.5	1,263
BSS-FC	0.10	16.8	25.5	1,515
BSS-FC	0.15	14.7	24.0	1,632
BSS-FC	0.20	13.1	21.4	1,634
BSS-FC	0.25	12.6	19.4	1,547
BTFNL	0.05	22.1	65.2	2,954
BTFNL	0.10	16.0	57.7	3,595
BTFNL	0.15	13.7	57.0	4,163
BTFNL	0.20	12.5	46.7	3,746
BTFNL	0.25	11.9	41.6	3,510

[5]The BSS Bloom filter targeted $\phi = 0.1$ for terms with IDF 4. The BSS-FC and BTFNL configurations set $\phi = 0.1$ for all terms, regardless of frequency.

Frequency consciousness reduces storage consumption while increasing speed. For example, at $d = 0.15$, the BSS configuration uses 46.7 bits per posting while the BSS-FC configuration uses only 14.7. This 3.2x reduction in storage is achieved while yielding a 2.6x increase in speed. The intuition behind the improvement is that frequency consciousness allows each term to have the right number of rows. With classical Bloom filters, every term has the same number of rows, meaning that more common terms get excess rows as a side effect of providing sufficient rows to ensure the target signal-to-noise level for rare terms.

Higher Rank Rows mainly impact speed. For example, when $d = 0.15$, BSS-FC runs at 24K queries per second, while BTFNL runs at 57.0K, a 2.4x improvement. The intuition behind the speed up is that higher rank rows can be scanned more quickly than rank-0 rows. Generally speaking, processing a rank-r row involves scanning $\frac{1}{2^r}$ of the quadwords necessary to process a rank-0 row.

The DQ column captures the tradeoff between space and speed. BSS-FC has a DQ of 1,632, while BTFNL has a DQ of 4,163, a 2.6x improvement. Combining frequency consciousness with higher rank rows yields a 21x improvement over that BSS DQ of 194.

We found that a density of 0.15 yielded the best DQ for Corpora B, C, and D, while A and E performed best at 0.05 and 0.20, respectively.

6.3 Comparison with Contemporary Indexes

The version of BitFunnel used by Bing includes a forward index with term frequencies used for BM25F ranking. Because this ranking code was not available to us at the time we designed our experiment, we limited our comparison to conjunctive boolean matching.

Our primary comparison system was Partitioned Elias-Fano or PEF[23]. This system is considered state-of-the-art, has excellent performance, and, like BitFunnel, is implemented in C++. We also compared with MG4J's Java implementation of PEF[6]. This implementation was the second fastest system in the SIGIR 2015 RIGOR workshop[18]. Our final comparison was with Lucene[7], a popular Java-based search engine that outperformed MG4J at the RIGOR workshop, in an apples-to-apples comparison using BM25F.

Each of these systems was configured to use a memory-mapped index that was non-positional, with scoring disabled. In this configuration, PEF and MG4J pay no runtime penalty associated with term frequencies because the frequencies are stored in a separate data structure that is never consulted. It is unclear whether Lucene pays a cost associated with stepping past term frequency values.

For each system we used 8 threads to process the entire 98k query log twice, back-to-back, measuring performance on the second pass. This ensured that relevant portions of the index were paged in, as they would be under continuous production load.

We can see from Table 3 that BitFunnel is faster than PEF in all cases, but sometimes this comes at a significant cost, for example in Corpus A, BitFunnel uses 5x as many bits per posting while yielding a false positive rate of 1.62%. Across the 5 corpora, MG4J is slower than PEF, as expected since it implements the same algorithm, but in Java. MG4J is faster than Lucene in all but Corpus C.

BitFunnel's overall performance relative to PEF improves as document lengths increase. It first surpasses PEF in Corpus C, where it

[6]http://mg4j.di.unimi.it/
[7]https://lucene.apache.org/

shows 3.2x the QPS of PEF while using only 2.6x the space. Examining DQ, the ratio of QPS to bits-per-posting, we see that BitFunnel outperforms PEF by factors of 1.3, 3.1, and 4.2 in Corpora C, D, and E, respectively, while PEF outperforms BitFunnel by factors of 3.4 and 1.6 in Corpora A and B.

Table 3: Query Processing Performance.

		BitFunnel	PEF	MG4J	Lucene
A	QPS	21,427	14,675	6,866	6,310
	False positives (%)	1.62	0.00	0.00	0.00
	Bits per posting	38.43	7.64	7.85	–
	DQ	558	1,921	875	–
B	QPS	8,674	5,049	3,636	3,011
	False positives (%)	4.32	0.00	0.00	0.00
	Bits per posting	20.72	7.33	7.59	–
	DQ	419	689	479	–
C	QPS	12,722	3,959	3,096	4,120
	False positives (%)	3.88	0.00	0.00	0.00
	Bits per posting	16.91	6.63	6.88	–
	DQ	752	598	450	–
D	QPS	57,014	8,268	5,900	3,632
	False positives (%)	2.43	0.00	0.00	0.00
	Bits per posting	13.69	6.25	6.28	–
	DQ	4,163	1,322	939	–
E	QPS	105,782	13,151	7,349	4,991
	False positives (%)	2.64	0.00	0.00	0.00
	Bits per posting	11.69	6.15	6.15	–
	DQ	9,047	2,139	1,195	–

These results are consistent with the interpretation that the biggest factor in BitFunnel performance is row length, which is directly proportional to the number of documents in the corpus. As document lengths increase and the corpus size drops, BitFunnel performance improves relative to PEF.

It is unclear from these results, the extent to which BitFunnel's performance gains are the result of a careful implementation versus actual algorithmic gains. We can see from PEF vs MG4J that choice of implementation language can have a significant impact on performance. Since BitFunnel compiles each query into x64 machine code, it is likely that some of BitFunnel's gains come from highly optimized query code.

7 CONCLUSION

This work revisits bit-sliced signatures and describes their use in a commercial search engine, which previously used inverted files. Signature-based approaches introduce several challenges and we develop a set of techniques to reduce the memory footprint and to process queries quickly. Furthermore, we derive a performance model that allows expressing the system configuration as an optimization problem. We evaluate the key techniques behind BitFunnel experimentally, and we provide the source code publicly to accelerate advances in this area.

8 ACKNOWLEDGMENTS

We thank the following colleagues for their contributions to BitFunnel: Andrija Antonijevic, Tanj Bennett, Denis Deyneko, Utkarsh Jain, and Fan Wang. We also thank the anonymous reviewers for their feedback which led to an improved experimental section.

REFERENCES

[1] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[2] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30, 1-7 (1998), 107–117.
[3] Jehoshua Bruck, Jie Gao, and Anxiao Jiang. 2006. Weighted bloom filter. In *2006 IEEE International Symposium on Information Theory*. IEEE.
[4] Stefan Büttcher, Charles LA Clarke, and Gordon V Cormack. 2016. *Information retrieval: Implementing and evaluating search engines*. Mit Press.
[5] Berkant Barla Cambazoglu and Ricardo A. Baeza-Yates. 2015. *Scalability Challenges in Web Search Engines*. Morgan & Claypool Publishers.
[6] J Shane Culpepper and Alistair Moffat. 2010. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)* 29, 1 (2010), 1.
[7] Bolin Ding and Arnd Christian König. 2011. Fast set intersection in memory. *Proceedings of the VLDB Endowment* 4, 4 (2011), 255–266.
[8] Chris Faloutsos. 1985. Access methods for text. *ACM Computing Surveys (CSUR)* 17, 1 (1985), 49–74.
[9] Christos Faloutsos. 1992. Information retrieval: data structures and algorithms. Prentice Hall PTR, 44–65.
[10] Christos Faloutsos and Stavros Christodoulakis. 1985. Design of a Signature File Method that Accounts for Non-Uniform Occurrence and Query Frequencies.. In *VLDB*. 165–170.
[11] Edward Fox, Donna Harman, w. Lee, and Ricardo Baeza-Yates. 1992. Information retrieval: data structures and algorithms. Prentice Hall PTR, 28–43.
[12] Shlomo Geva and Christopher M De Vries. 2011. Topsig: Topology preserving document signatures. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 333–338.
[13] Andrew Kane and Frank Wm Tompa. 2014. Skewed partial bitvectors for list intersection. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. ACM, 263–272.
[14] A Kent, Ron Sacks-Davis, and Kotagiri Ramamohanarao. 1990. A signature file scheme based on multiple organizations for indexing very large text databases. *Journal of the American Society for Information Science* 41, 7 (1990), 508.
[15] Donald E Knuth. 1998. *The Art of Computer Programming, Vol. 3, Sorting and Searching* (2nd ed.). Vol. 3. Addison-Wesley, 567–573.
[16] Roberto Konow, Gonzalo Navarro, Charles LA Clarke, and Alejandro López-Ortíz. 2013. Faster and smaller inverted indices with treaps. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 193–202.
[17] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.
[18] Jimmy Lin, Matt Crane, Andrew Trotman, Jamie Callan, Ishan Chattopadhyaya, John Foley, Grant Ingersoll, Craig Macdonald, and Sebastiano Vigna. 2016. Toward reproducible baselines: The open-source ir reproducibility challenge. In *European Conference on Information Retrieval*. Springer, 408–420.
[19] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. 2001. Building a distributed full-text index for the Web. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*. 396–406.
[20] Alistair Moffat and Justin Zobel. 1996. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems (TOIS)* 14, 4 (1996), 349–379.
[21] Calvin N Mooers. 1948. *Application of random codes to the gathering of statistical information*. Ph.D. Dissertation. Massachusetts Institute of Technology.
[22] Calvin N Mooers. 1951. Zatocoding applied to mechanical organization of knowledge. *American documentation* 2, 1 (1951), 20–32.
[23] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned elias-fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. ACM, 273–282.
[24] Knut Magne Risvik, Trishul M. Chilimbi, Henry Tan, Karthik Kalyanaraman, and Chris Anderson. 2013. Maguro, a system for indexing and searching over very large text collections. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*. 727–736.
[25] Charles S Roberts. 1979. Partial-match retrieval via the method of superimposed codes. *Proc. IEEE* 67, 12 (1979), 1624–1642.
[26] Ron Sacks-Davis, A Kent, and Kotagiri Ramamohanarao. 1987. Multikey access methods based on superimposed coding techniques. *ACM Transactions on Database Systems (TODS)* 12, 4 (1987), 655–696.
[27] Harry KT Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. 1985. Bit Transposed Files.. In *VLDB*, Vol. 85. Citeseer, 448–457.
[28] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. 1998. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)* 23, 4 (1998), 453–490.