Technical Report

Inverted List Structure for High Performance Common Set Retrieval : Prototype Performance and Algorithmic Details

Peter Williams
September 14th 2012

I. Abstract

This document describes the prototypes developed to improve SF1R query latency and single server capacity, and proof-of-concept performance tests.

There are two inverted list data structures described here. Firstly, a bitmap-based approach which gives exceptional latency performance, but in general is inefficient in memory usage. Query latency for an AND query of two inverted lists with a common set of size 1 million documents— the likely size of large common set results computed from a corpus of 1 billion documents —was found to be < 0.004 seconds, scaling linearly with the size of the common set and number of documents in the corpus.

The second approach is a very efficient vectorized implementation of a traditional plain representation of an inverted list using 4-byte document ids. In practical situations, the latency is > 2 - 10 times slower than the bitmap approach, but the data structures are at least 10 times more efficient in memory usage.

Both implementations depend heavily on low-level coding techniques: multi-threading, vectorization, assembly language, memory caching, and bitmaps, to achieve high performance. These aspects are discussed in detail below.

The final solution suggested here is a highly configurable hybrid approach, which exploits the strengths of both approaches to give optimal overall performance. That is: inverted lists of the most frequently used tokens are stored in bitmap format and cached in memory, giving the best latency possible for the most common queries; less common tokens have their inverted list data stored in the second format, either memory cached or stored on disk. Inverted list data in bitmap format can be efficiently converted to the second inverted list data structure on the fly, allowing for queries which include tokens having both inverted list types.

In addition, for large document corpuses, two separate indices are computed: a pruned index containing the highest ranking documents, likely to satisfy the most popular queries; and a full index containing all documents. This index pruning approach gives optimal performance for popular queries, while still allowing full document recovery for rare queries.

In conclusion, this new approach to common set retrieval has the potential to give many orders of magnitude in performance improvement on open source search technology and the current SF1R codebase; and would provide a significant product differentiation and technological advantage for izenesoft.com and b5m.com.

I. Algorithm overview in pseudo code

Before stepping into the details, it is useful to set out the algorithm in its major steps in simple terms. The input is a json query, in the format currently used by SF1R. The output is the common set results in SF1R json format.

```
* tokenize raw query into token ids;

* write query logic and operators in reverse polish notation;

if ( query common set is cached )
{
    common set result = cached common set;
}
else
{
    first use pruned inverted lists;

    start_computation:

    if( there are any query tokens with inverted lists not cached in memory )
    {
        read them from disk cache;
    }

    call thread code (pseudo code given below) to compute partial common sets in parallel;

    combine partial common sets into final common set result;

    if( results set is too small and we have not already checked the full index )
    {
        use full inverted lists;

        goto start_computation;
    }
}

* retrieve requested data fields for these document ids, construct json response;
```

Note that the tasks marked with an asterisk * are not considered in this document. Here is the thread pseudo code:

```
common set result = inverted list of first token

for each( remaining token, query operator )
{
    common set result = operator( common set result, next token );
}

filter common set result by query attributes filter;

find top n results ranked by a score;
```

II. Inverted List Data Structure

In general terms, an inverted list is a mapping between a token and a set of document ids. The inverted list data structures described here have a number of additional special properties which help improve performance.

1. Optimal internal document labelling to reduce sorting

The internal labelling of documents with ids is something we are free to do as we like. The only requirement is that each document has a unique id which can be easily mapped to an external document id.

During indexing, documents are ordered by a default ranking score, and re-labeled with their ranking (an unsigned integer). Each inverted list (i.e. each set of document ids) is then sorted into numerical order. These unsigned integers are used as the internal document id.

In this way, we can avoid the need for any sorting when computing common sets using the default ranking: the lowest n document ids in the common set have the highest n ranking scores. Also, the fact that the inverted lists are sorted by document id helps in the implementation of the common set computation (see following sections).

A simple example demonstrating these concepts is given below, where there are just 3 documents constructed from a dictionary of 4 tokens:

external document id (a string)	default score	internal document id (= score ranking)	token ids
“d00”	0.2	2	1 3 2
“d01”	1.0	0	0 1 2
“d02”	0.3	1	0 1

The resultant inverted lists are:

token id	internal document id	external document ids (in the same order)
0	0 1	“d01” “d02”
1	0 1 2	“d01” “d02” “d00”
2	0 2	“d01” “d00”
3	2	“d00”

Consider a query for token ids 1 AND 2. Taking the above inverted lists, this becomes (0 1 2) AND (0 2) = (0 2) = (“d01” “d00”). Thus the common set is already sorted by score (scores 1.0 and 0.2 respectively).

2. Inverted list structures to enable multi-threaded parallel common set computation

Given N documents which are numbered 0 to N - 1 and n_t threads, data is shared amongst threads by sharing the range of document ids evenly amongst the threads, i.e. documents from id = i N / n_t to id = (i + 1) N / n_t are assigned to the ith thread (threads are numbered starting at zero).

The job of each thread is to find the common set results which lie in the document id range assigned to that thread. The output for each thread is the top n_k results for the input query. A parent thread then combines the top results from each thread, finding the top n_k results across all threads, a negligible amount of additional work.

In summary, the inverted list structure used here is an ordered set of 4-byte document ids, segmented into n_t parts. For example, consider the case N = 16 and n_t = 2. For the inverted list (0, 1, 5, 6, 7, 9, 11, 12, 15), thread 0 computes document ids $0 \leq id < 8$, i.e. the first 5 document ids; and thread 1 computes document ids $8 \leq id < 16$, i.e. from the 6th document id to the end.

3. Inverted list structure no. 1 : bitmap based

At the highest level, the purpose of an index is to indicate whether or not a word is present in a document. (We will deal with document fields below.) Thus, for a document corpus of size N, an inverted list could be represented as a bitmap of length N bits. But clearly this does not scale well for high N and large numbers of tokens because most inverted list bitmaps are very sparse.

This wastefulness can be avoided by partitioning the document ids into blocks of equal size, in the same way that the document ids were partitioned for multithreading. For each token, we can then construct a bitmap which indicates which document id blocks contain non-zero bits.

These are the two main elements of the inverted list structure. For each token, a block id bitmap, and the corresponding document id bitmaps for each non-empty document id block. Common set computation is then carried out using binary operations to combine these bitmaps. Note that these operations can be easily vectorized using Intel Intrinsics (see section IX. for a short review of vectorization). With 128-bit registers, a minimum specification for Intel chips, 128 document ids can be compared with just one Intrinsics operation, equivalent to a small number of assembly language commands.

An example will make the elements of the algorithm clearer. Let us imagine a case where there are N = 16 documents in a corpus, and token A is found in documents 0, 4, 5, and 15. The inverted list for token A can be represented as [1000, 1100, 0000, 0001]. Note that we do not need to store the third zero bitmap in a real implementation. For simplicity lets assume that we are using 4 document id blocks. Then the block id bitmap for this token is 1101.

Imagine we have another token B with block id bitmap 1001, and document id bitmaps of [0010, 0000, 0000, 0001]. Clearly tokens A and B have document id 15 in common.

To compute the common set A&B, first we compute using the block id bitmaps: 1101 & 1001 = 1001. This result tells us tokens A and B both have document ids in the blocks 0 and 3.

The next step is to compute the AND operation for the first block from token A and B: 1000 & 0010 = 0000. There are no documents in common in this range.

Next, we carry out the same operations for the third non-zero document id block from A, and the second non-zero document id block from B: 0001 & 0001 = 0001. This shows that document 3 of block 3 has a document in common, i.e. document id = 3 x block size (= 4) + 3 = 15. Thus we have found the common set with just a small number of binary operations on the bitmaps.

4. Inverted list structure no. 2 : traditional 4-byte document id approach with vectorization

The next segmentation of the inverted lists by document id range is to enable vector computation of the common set. For n_r registers, the jth register on the ith thread is assigned the documents with id = $(N / n_t) \times (i + j / n_r)$ to id = $(N / n_t) \times (i + (j+1) / n_r)$.

However, for effective vectorization, we need the values to be computed in parallel to be contiguous in memory. Unlike the thread-segmentation of the inverted list, we need to interleave the segmented values for each register. This can be more clearly understood with an example. For simplicity, we take N = 16, $n_r = 2$, and $n_t = 2$ in this example. Thus the document ids covered by each thread and register are:

thread id	register id	document id range
0	0	0, 1, 2, 3
0	1	4, 5, 6, 7
1	0	8, 9, 10, 11
1	1	12, 13, 14, 15

Consider the example raw inverted list: (0, 1, 5, 6, 7, 9, 11, 12, 15). Using our multithreading inverted list data segmentation rule, thread 0 works with the inverted list (0, 1, 5, 6, 7) and thread 1 works with the inverted list (9, 11, 12, 15).

Since there are two registers, for thread 0, register 0 works with the document ids (0, 1) and register 1 has document ids (5, 6, 7).

In memory, these values will be ordered: (0, 5, 1, 6, null, 7), where the document ids computed by register 0 are underlined. In the case where one inverted list is shorter, the values will be padded, indicated with a null here.

When moving through the inverted list, the document ids within these square brackets: [0, 5] , [1, 6] , [null, 7] can be loaded into registers in one operation and computed on simultaneously.

The same logic can be followed through for thread 1's inverted list. Thus the raw inverted list (0, 1, 5, 6, 7, 9, 11, 12, 15), when written to disk during indexing, will have the ordering (0, 5, 1, 6, null, 7, **9, 12, 11, 15**). Register 0 document ids are shown underlined, and thread 1's inverted list is show in bold.

III. Prototype benchmarking results

The benchmarking data is incomplete, although I have enough data to assess the rough performance of the two inverted list data structures and additional operations.

These tests were carried out on a Macbook Pro with a 4-core Intel i7 processor which has four 32-bit registers for vectorization. Inverted lists for two tokens were generated randomly, with some fixed fraction of document ids in common. Timing results were averaged over 100 randomly generated cases, and for each generated instance query retrieval was repeated 10,000 times.

1. The bitmap based inverted list structure

For the bitmap-based inverted list structure, the key performance statistic is the time taken per bitmap comparison. This was measured as 8×10^{-8} seconds. An upper bound on the number of bitmap comparisons required to compute a common set is

$$\sim 8 \times 10^6 \times (N / 10^9) \times (N_c / 10^7) \times (10^7 / N_b) / f \quad \text{bitmap comparisons,}$$

where N_b the number of block ids, N_c the number of results in the common set, and f is the average number of bits filled per 128 register, $1 \leq f \leq 128$, dependent on the data. Typical values are used. (This and the results below are derived in section VIII.) Using the timing for a bitmap comparison, this can be converted into worst-case latency of

$$\sim 0.04 \times (N / 10^9) \times (N_c / 10^7) \times (10^7 / N_b) \times (16 / n_t) / f \quad \text{seconds,}$$

where n_t is the number of cores (the test server we have has 16 cores). From this equation we can see that even for data sets as large as 1 billion documents and large common sets, the latency is impressive. Note that latency per common set value is lower when f is high, i.e. for tokens which have many document hits.

However, this data structure is not memory efficient. The memory size of the complete index is

$$\sim 12 \times (N_b / 10^7) \times (N_t / 10^7) + 0.01 \times (N_w / 10^2) \times (N / 10^9) \quad \text{Tb,}$$

where N_t is the number of tokens in the dictionary and N_w the average number of unique token ids per document. The first term is the memory taken for block id bitmaps, the second for the document id bitmaps.

2. Four-byte document id inverted list structure

Memory requirements for this inverted list structure are

$$\sim 0.4 \times (N_w / 10^2) \times (N / 10^9) \text{ Tb.}$$

this is more memory efficient than the worst case bitmap based inverted list structure. However, the data structure does not allow the same efficiency in computation: the latency for this algorithm scales as

$$\sim 0.06 \times (N_i / 10^8) \times (16 / n_t) \quad \text{seconds,}$$

where N_i is the number of document ids per inverted list. Because the algorithm scalings depend on different parameters, it is a little difficult to make a direct comparison. Although the values used in the two latency formulas were chosen to give as fair-as-possible a comparison. Clearly even the worst-case bitmap latency is faster, and since the factor f may be 10 or higher for some tokens, the bitmap approach may be > 10 times faster for some inverted lists.

3. Timings of remaining operations

There are a number of other operational steps needed to return the common set for a given query (see the pseudo code in section I.). Firstly, retrieval of disk-cached inverted lists from disk. On a MacBook Pro, the disk reading speed was found to be

$$\sim 0.8 \times (N_i / 10^8) / f \text{ seconds (bitmap structure), and}$$

$$\sim 0.2 \times (N_i / 10^8) \text{ seconds (4-byte structure).}$$

This implies that when inverted lists are read from disk, this will be the most time consuming task.

The tasks “combine partial common sets into final common set result”, “filter common set result by query attributes filter”, and “find top n results ranked by a score” were benchmarked to take < 10% of the common set retrieval times. The data structures and algorithms related to these steps are discussed below in sections V. and VI.

IV. Conclusions

The results of this prototyping of two inverted list data structures has shown that:

1. by considering carefully the six aspects mentioned in section I: multi-threading, vectorization, assembly language, caching, document pruning, and coding style, high performance search can be achieved, even on desktop hardware.
2. the bitmap-based inverted list structure has the potential to give very impressive latency performance, however it has poor memory efficiency for inverted lists containing few hits.
3. The 4-byte document id approach provides reasonable latency performance for small to medium-sized data sets, and is comparatively memory efficient.
4. The natural conclusion is to combine both approaches, allowing some tokens to be stored in bitmap-format, and others in 4-byte format. A latency-focused approach would involve memory caching all popular tokens in bitmap format, with the full inverted lists cached on disk in 4-byte format. A memory-usage focused approach would involve computing high-frequency token inverted lists in bitmap format, and low frequency token inverted lists in 4-byte format. The optimal choice is data dependent, and thus it is preferable that both data structures are provided, and are configurable to suit any particular data set.
5. One negative aspect of using vectorization approaches such as Intrinsics is that

the code is not portable across chip makers. The codes written here cover Intel and AMD chips, covering the majority of processors we will encounter. This minor sacrifice in code portability may be worth the performance gains.

Other aspects of interest in the design presented here are:

1. document ids and document data (attributes, text fields) are stored as separate objects in memory. This allows for real time updating of any of these fields (eg. product price, category) without affecting search performance.
2. for index updates which involve adding or deleting documents, the bitmap-based inverted lists are most efficient: such updates involve nothing more than binary & operations on the document id bitmaps, and possibly the addition of a block id. Such updates are possible too with the 4-byte data structure, although the code will be more complex. Alternatively, document updates may be better handled by making a small document-based index of this new data, computing common sets from the original index and this new index, and combining common set results.
3. There are many ways to improve performance beyond what I have reported here. Firstly, dedicated servers can be expected to be more powerful than a MacBook Pro, having better caching, and higher clock rate. Also, there is the possibility of using processors which have more cores, and larger registers. An increase to 32 cores and 256 bit registers would give a speedup of 4 on the numbers given here. Secondly, these prototype tests were carried out using the open source gnu c++ compiler. But some benchmarking tests show that the Intel c++ compiler may perform better, depending on the problem (see the table at the bottom of this link). Some experimentation with different compilers, and a thorough exploration of compiler options may give significant further improvements in performance. Beyond these improvements, we are left with a distributed implementation to increase capacity and improve performance further. From the above discussion of parallelisation using threads and registers, it is clear that distributed parallelisation is logically the same design pattern, and the implementation will be structured in a manner which allows easy implementation of a distributed version in future.

V. Document Attributes Data Structure

In the algorithm discussed above, inverted list data structures contain only document ids. All other data related to the documents are stored elsewhere in memory. This is primarily so that the memory map of inverted list document ids is optimized for performance, as discussed above. But this approach also means that it is easy to update document attributes in real time (such as product price), without having to change the inverted lists in any way.

The allowed document attributes are of four types: string, single-valued float, single-valued integer, and multiple-valued integer.

String attributes are stored without compression. Their use will be rare, but cover cases where there is a text attribute associated with a document which we do not want indexed. For example, external document ids may be stored as a string attribute. Single valued float attributes, for example product lengths or heights in metres, are stored as 4 byte floats. Other attribute value types are compressed, as discussed below.

Single valued integer attributes are the preferred data format for attributes, because of their memory efficiency. Attributes that can be stored in this way are for example, category, brand or price. Note that external to the common set retrieval, attribute values may be associated with strings such as brand, or they may just be numbers like price. But internally, all attributes are reduced to integers. Moreover, they are represented with only the number of bits required. For example, let us imagine we have 700 brands. The value 0 is reserved to indicate that the attribute is not set. Thus we need to be able to represent 701 possible values for the brand attribute, which requires $1 + \log_2(701) = 10$ bits per product.

In practice, float attributes can often be reduced to an integer attribute with no loss of freedom. For example, float value lengths in metres can be converted to integer millimetres. This is the preferred method, since the storage of integer attributes is much more memory efficient. Other more complex attributes are often required, such as price ranges. These should also be implemented as single-valued integer attributes.

Multi-valued attributes require n bits for an attribute which can take n values. For example, a product may have two or more colours selected from a fixed palette of 20 colours. The size of this attribute per product is 20 bits.

Due to the possibly large number of values, sometimes multi-valued attributes can be implemented more efficiently by fixing a maximum number of values that can be set. For example, if we decide that a product can have no more than 3 colours from a palette of 20, then we can implement this colour attribute as three $1+\log_2(21) = 5$ bit single-value integer attributes: that is 15 bits per product, in comparison to 20 bits for the full multi-value attribute.

Attribute filtering involves taking a logical AND of the input attribute filter bitmap, set by the query, with the attribute bitmaps of the products. This is a very efficient operation, which can also be vectorized with SIMD calls.

VI. Field Hits and Token Positions Data Structure

In the previous sections we have discussed inverted lists, which tell us which documents possess a given token, and also how documents are filtered out of the common set by attribute combinations.

A document may consist of a number of text fields, e.g. title and description, and we may also may want to find documents with hits only in some given fields. Field hits filtering is implemented as a multi-valued attribute, as described above. For example, for documents with three text fields, we need a 4-bit multi-valued attribute.

There may also be cases where we need to know the positions of the token hits within a field, for use in the scoring function.

The document text fields represented as token ids are used to retrieve the token position data by simply running through the field token ids for each document id in the common set. Since web search document fields are usually < 100 tokens, e.g. product titles or descriptions, this is a reasonably efficient way to retrieve this data.

This approach is also memory efficient, and can also be considered as a text compression method. There are on average 1.6 characters per word in Chinese, and utf8 Chinese characters use 3 bytes each. Thus on average, n bytes of utf8 text contains ~ n / 4.8 tokens. The number of tokens in the dictionary will be ~ 10^7: $\log_2 (10^7)$ ~ 24 bits ~ 3 bytes are required to represent all token ids. Thus n bytes of utf8 Chinese text can be compressed to ~ 3 x n / 4.8 bytes ~ 0.6 n bytes using this method.

Further compression gains may be achieved by compressing the document field token ids by document. But this is left for future work, if further capacity increases are required.

VII. Derivation of the memory and latency scaling estimates

Operation counts for a common set search involve N_b / 128 bitmap comparisons using SIMD operations, where N_b is the number of block ids.

If N_c is the number of document ids in the resultant common set, the number of document id bitmap comparisons needed is

$$(N_c / f) \times (N / 128 N_b) \text{ operations,}$$

where f is the average number of bits filled per 128 register, $1 \leq f \leq 128$, dependent on the data, and N is the total number of documents. The first term is the number of 128 bit registers which need to be checked to find the N_c common set document ids. The second term is the number of 128 bit register values within the range of document ids covered by one block id.

By looking at reasonable values for these parameters, we can see that the block id computational cost is always much less than the document id bitmap comparisons. Thus we can estimate the computational cost as

$$\sim N_c N / 128 f N_b \text{ operations,}$$

The memory taken up by the block id bitmaps is

$$N_t \; N_b \; / \; 8 \text{ bytes,}$$

where N_t is the number of tokens in the dictionary.

Memory estimations for the document id bitmaps are more complex, because the size depends on the characteristics of the data. But we can get some reasonable estimates. If N_w is the average number of unique token ids per document, the fraction of all document ids with hits is $N_w \; N \; / \; N_t \; N$. Thus, on average $N_b \; x \; N_w \, / \, N_t$ document id bitmaps will contain non-zero bits, for each inverted list. Each document id bitmap is of size $N \; / \; 8 \; N_b$ bytes.

Bringing all of these results together, the total memory size is

$$\sim N_t \; N_b \; / \; 8 + (N_b \; x \; N_w \, / \, N_t \;) \; x \; N \; / \; 8 \; N_b$$

Rearranging, this becomes

$$\sim N_t \; N_b \; / \; 8 + N \; x \; N_w \; / \; 8 \; N_t \; .$$

IX. Vectorization and assembly language

Vectorization is the parallel computation of simple operations at the lowest level of the CPU. The latest Intel chips provide an interface for programmers to use the Single Instruction Multiple Data (SIMD) vectorization capability of their chips, which all have at least 4 channels, each of 32 bits, and 8 channels on some chips. This means that operations can be carried out on 4 (or 8) 4-byte words simultaneously, speeding up performance theoretically by 4 (or 8) times.

Traditionally, vectorization required the programmer to write assembly language, which is not portable and difficult to debug, and such functionality was only available on high-end super computers. However, to answer ever increasing video and sound demands principally from the gaming industry, Intel now provide wrapper functions for commonly needed SIMD operations, called Intrinsics, which give performance nearly as good as pure assembly language but with added portability. Moreover, the newest versions of the open source gnu c and c++ compiler also support these method calls.

To get a feel for what programming with Intrinsics looks like, a simple example is given below which computes the sum c = a + b, where a, b, and c are 4-dimensional vectors of 4-byte floats. The traditional code is:

```
void add(float *a, float *b, float *c)
{
        for ( int i = 0; i < 4; i++) c[i] = a[i] + b[i];
}
```

Using Intel intrinsics, the code becomes[1]

1 For reference, the assembly language version looks something like:

```
void add(float *a, float *b, float *c)
{
        __m128 t0, t1;                // instance 2 128-byte register variables
        t0 = _mm_load_ps(a);         // load 4 x 4 byte floats into a register from *a
        t1 = _mm_load_ps(b);         // load 4 x 4 byte floats into a register from *b
        t0 = _mm_add_ps(t0, t1); // add them in parallel, with one operation
        _mm_store_ps(c, t0);          // copy result to memory *c
}
```

In tests carried out on a MacBook Pro (four 32-bit registers), the Intrinsics version performed ~4.8 times faster than the traditional code. This exceeds the expected factor of 4 probably because the Intrinsics version also saves on the need for loop counter variables.

```
void add(float *a, float *b, float *c)
{
        __asm{
                mov eax, a
                mov edx, b
                mov ecx, c
                movaps xmm0, XMMWORD PTR [eax]
                addps xmm0, XMMWORD PTR [edx]
                movaps XMMWORD PTR [ecx], xmm0
        }
}
```

clearly less understandable than the Intrinsics example.