博客
新闻
工程
用户案例
发布
档案
RSS

2015年2月18日 Engineering

Frame of Reference and Roaring Bitmaps

作者
Adrien Grand

Share

Postings lists

While it may surprise you if you are new to search engine internals, one of the most important building blocks of a search engine is the ability to efficiently compress and quickly decode sorted lists of integers. Why is this useful? As you may know, Elasticsearch shards, **which are Lucene indices under the hood**, split the data that they store into segments which are regularly merged together. Inside each segment, documents are given an identifier between 0 and the number of documents in the segment (up to 2^{31}-1). This is conceptually like an index in an array: it is stored nowhere but is enough to identity an item. Segments store data about documents sequentially, and a doc ID is the index of a document in a segment. So the first document in a segment would have a doc ID of 0, the second 1, etc. until the last document, which has a doc ID equal to the total number of documents in the segment minus one.

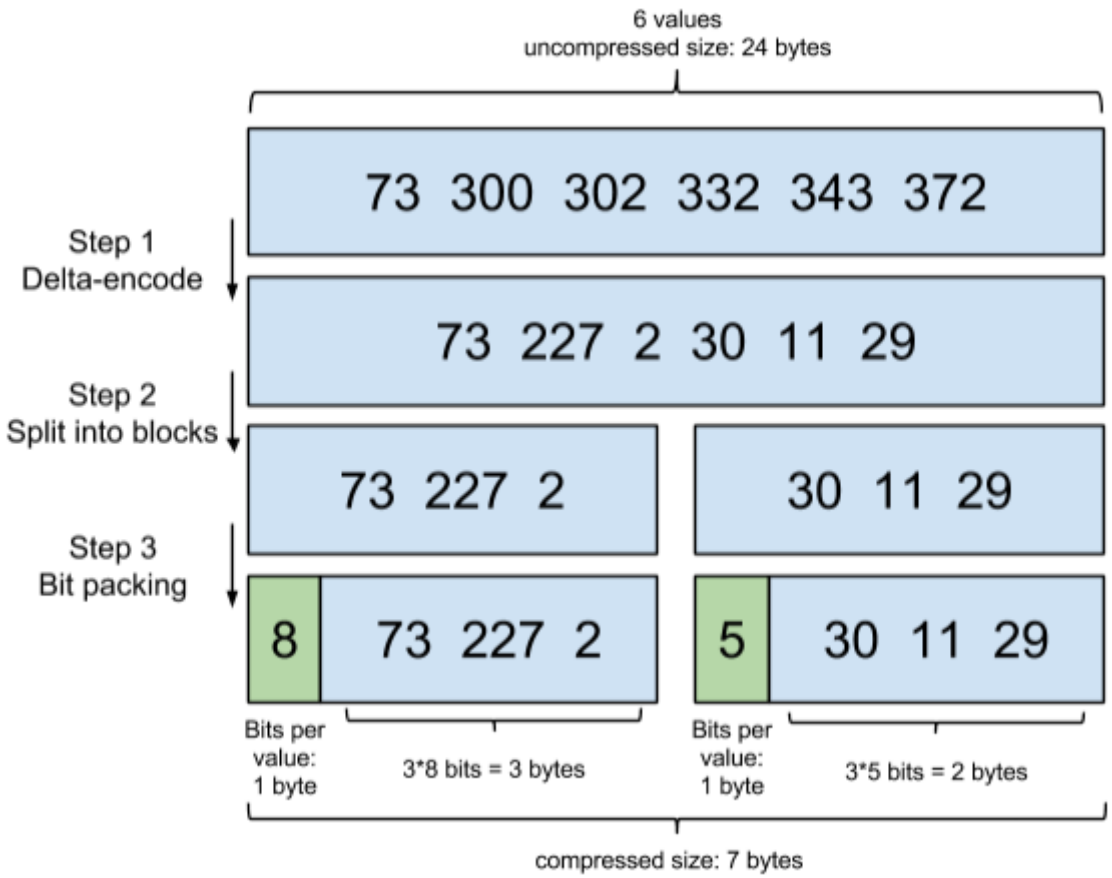
Why are these doc IDs useful? An inverted index needs to map terms to the list of documents that contain this term, called a *postings list*, and these doc IDs that we just discussed are a perfect fit since they can be compressed efficiently.

Frame Of Reference

In order to be able to compute intersections and unions efficiently, we require that these postings lists are sorted. A nice side-effect of this decision is that postings lists can be compressed with delta-encoding.

For instance, if your postings list is `[73, 300, 302, 332, 343, 372]`, the list of deltas would be `[73, 227, 2, 30, 11, 29]`. What is interesting to note here is that all deltas are between 0 and 255, so you only need one byte per value. This is the technique that Lucene is using in order to encode your inverted index on disk: postings lists are split into blocks of 256 doc IDs and then each block is compressed separately using delta-encoding and bit packing: Lucene computes the maximum number of bits required to store deltas in a block, adds this information to the block header, and then encodes all deltas of the block using this number of bits. This encoding technique is known as *Frame Of Reference* (FOR) in the literature and has been used **since Lucene 4.1**.

Here is an example with a block size of 3 (instead of 256 in practice):



The same abstraction is used at search time: queries and filters return a sorted iterator over the list of documents that they match. In the case of term queries and filters, implementation is very simple, we just need to return an iterator over a postings list from the inverted index. Other queries are more sophisticated. For instance, a disjunction `termA OR termB` would need to merge postings lists for `termA` and `termB` on the fly. But in the end, it is still using the same abstraction.

Roaring bitmaps

This leads us to a second place where Lucene needs to encode sorted lists of integers: the filter cache. Filter caching is a popular technique which can speed up the execution of frequently-used filters. It is a simple cache that maps (filter, segment) pairs to the list of doc IDs that they match. But constraints are different from the inverted index:

- Since we only cache frequently-used filters, the compression ratio does not

matter as much as for the inverted index which needs to encode matching documents for _every_ possible term.

- However, we need cached filters to be faster than re-executing the filter, so it is important to use a good data-structure.
- Cached filters are stored in memory while postings lists are typically stored on disk.

For these reasons, the best encoding techniques are not necessarily the same for an inverted index and for cached filters.

So what should we use here? Clearly the most important requirement is to have something fast: if your cached filter is slower than executing the filter again, it is not only consuming memory but also making your queries slower. The more sophisticated an encoding is, the more likely it is to slow down encoding and decoding because of the increased CPU usage, so let's look at the simple options that we have to encode a sorted list of integers:

Option 1: integer array

Probably the simplest option: doc IDs are stored in an array. This makes iteration very simple, however compression is really bad. This encoding technique requires 4 bytes per entry, which makes dense filters very memory-consuming. If you have a segment that contains 100M documents, and a filter which matches most documents, caching a single filter on this segment requires roughly 400MB of memory. Ideally we should have something more memory-efficient on dense sets.

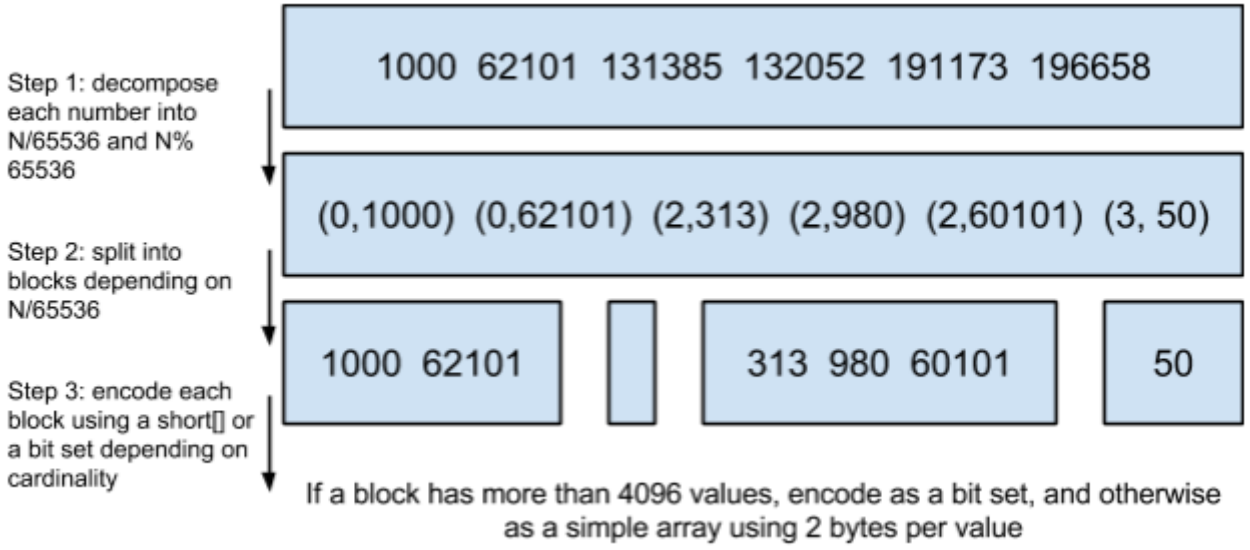
Option 2: bitmap

Dense sets of integers are a great use-case for bitmaps. A bitmap is an array where each entry takes only one bit, so they only have two possible values: 0 or 1. In order to know whether docID is contained in a bitmap, you need to read the value at index `docID`. 0 would mean that the set does not contain this docID while 1 would mean that the set contains this docID. Iteration requires to count consecutive zeros, which is actually very fast since CPUs have dedicated instructions for that. If we compare to option 1, memory usage is much better on dense filters since we would now only need 100M bits = 12.5MB. But now we have an issue with sparse sets: while we needed 4 bytes per match with our first option, we now need 12.5MB of memory, no matter how many matches there are.

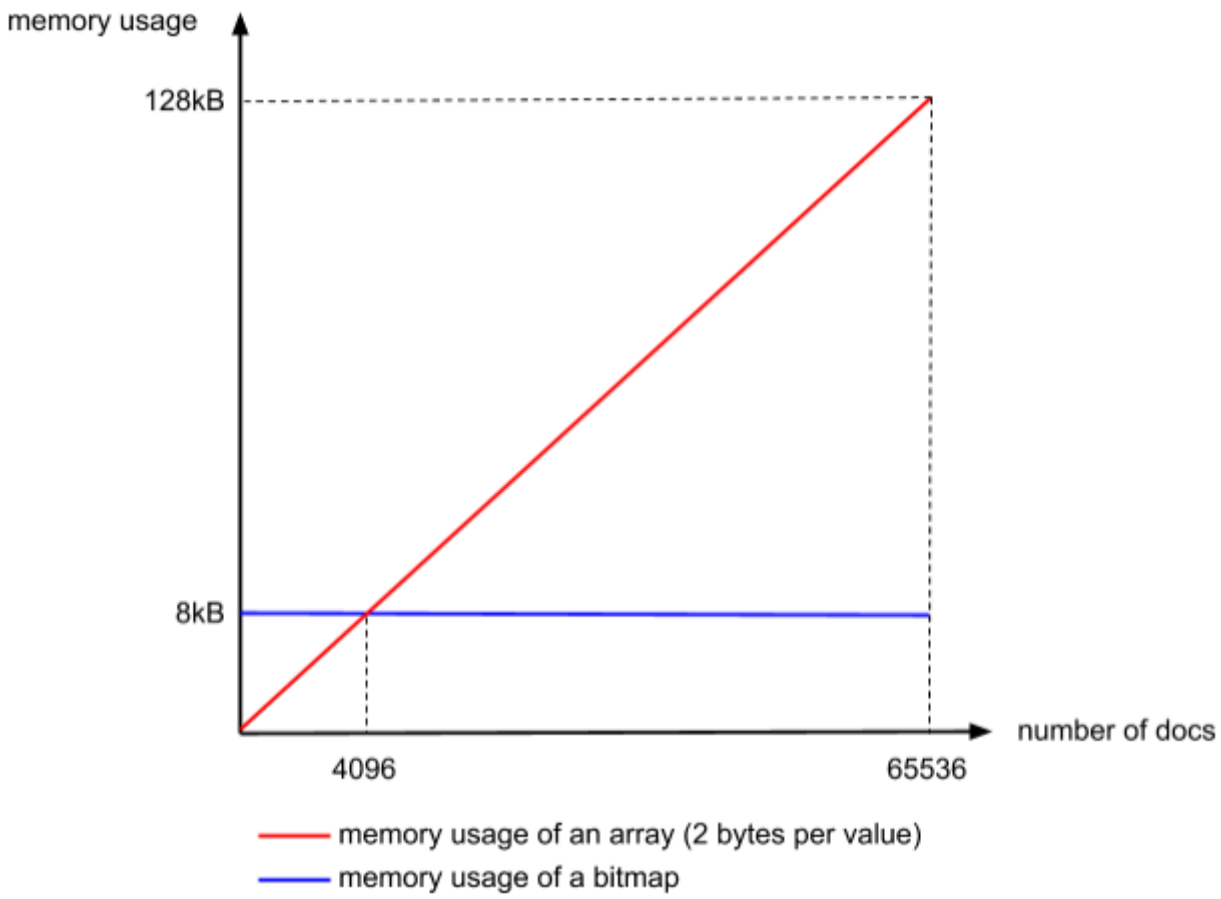
For a very long time, Lucene has been using such bitmaps in order to cache filters into memory. In Lucene 5 however, we switched to Daniel Lemire's **roaring bitmaps**. See **LUCENE-5983** for more background.

Option 3: roaring bitmaps

Roaring bitmaps aim at taking the best of both worlds that we just described. It starts by splitting the postings list into blocks depending on the 16 highest bits. Which means that for example, the first block would encode values between 0 and 65535, the second block between 65536 and 131071, etc. Then in each block we encode independently the 16 lowest bits: if it has less than 4096 values, an array will be used, otherwise a bitmap. Something important to notice at this stage is that while we used to need 4 bytes per value with the array encoding described above, here the arrays only need to store 2 bytes per value since the block ID implicitely gives us the 16 highest bits.



Why does it use 4096 as a threshold? Simply because above this number of documents in a block, a bitmap becomes more memory-efficient than an array:



This is what makes roaring bitmaps interesting: they are based on two fast encoding techniques that have very different compression characteristics and dynamically decide which to use based on memory-efficiency.

Roaring bit maps have lots of features, but there are really only two of them that interest us in the context of Lucene:

- Iterating over all matching documents. This will typically be used if you run a **constant_score** query over a cached filter.
- Advancing to the first doc ID contained in the set which is greater than or equal to a given integer. This will typically be used if you **intersect the filter with a query**.

Comparison

Let's compare several **DocIdSet** implementations to see why we decided on using roaring bitmaps for filter caching. Here are the various implementations that we are

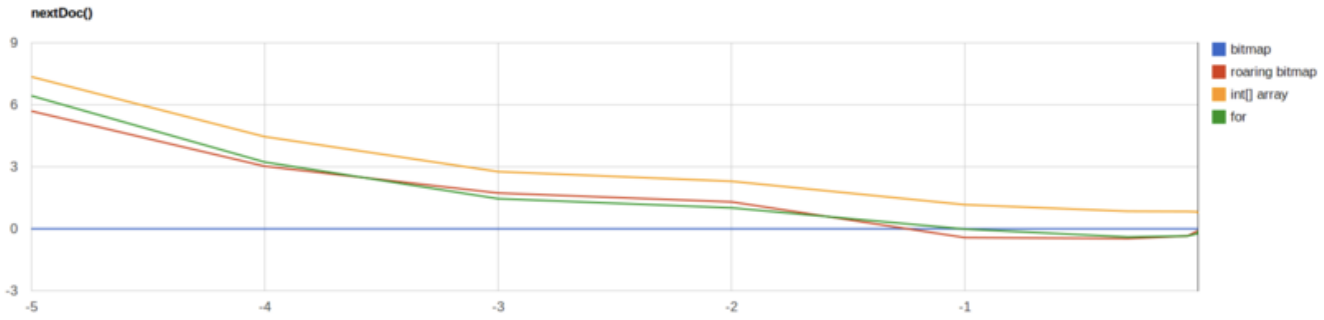
comparing:

- int[] array: see option 1 above, using this implementation.
- bitmap: see option 2 above, using Lucene's BitDocIdSet over a FixedBitSet.
- roaring bitmap: see option 3 above, using Lucene's RoaringDocIdSet.
- for: a disk-based postings list as created by Lucene during indexing, in the OS cache at the moment of the benchmark.

The benchmark code is available at https://code.google.com/a/apache-extras.org/p/luceneutil/source/browse/src/main/DocIdSetBenchmark.java

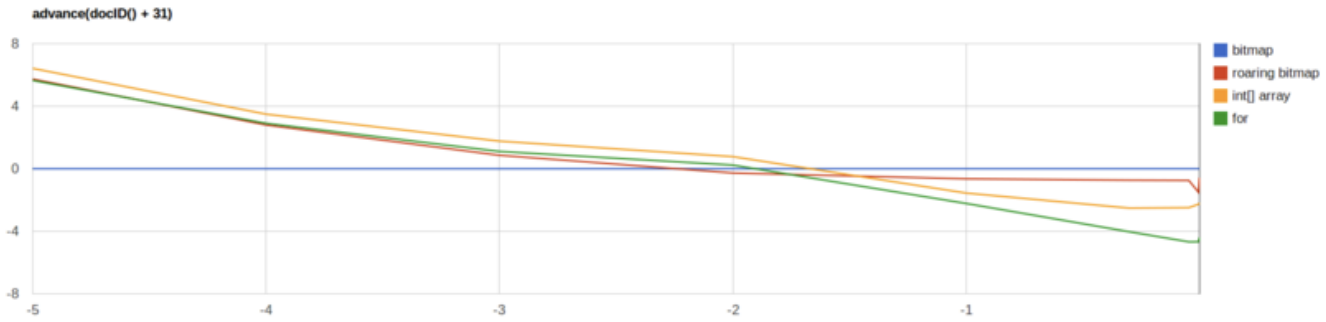
In all the charts that will be presented, we use bitmap as a reference implementation since it had been used for years in Lucene. The y-axis uses a logarithmic scale in base 2: a value of 0 means that it is as fast as a bitmap, 1 means 2x faster, etc. The x-axis uses a logarithmic scale in base 10 which represents the density of the doc id set. For instance a value of -2 means that 10^{-2}=1% of documents are contained in the set.

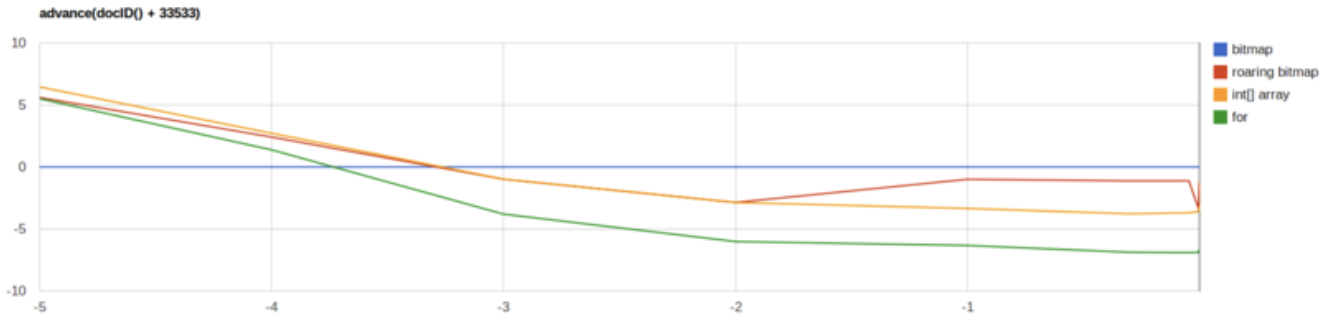
Iteration performance



Here we are measuring iteration performance, which is essentially about the performance that you would get when wrapping the filter in a constant-score query. The int[] array constantly beats other implementations by a factor of 2. What is even more interesting to notice is that the bitmap is much slower than other implementations in the sparse case, which means that you should not use it to cache filters as it could be even slower than reading from disk again.

Skip performance





This time we are measuring skipping, which is used when you intersect a filter with another query. The number in parenthesis is the number of documents that we are trying to skip over (no matter whether they match or not) on every iteration. Basically skipping over N documents would occur when intersecting with a query that matches roughly 1/Nth of all your documents.
You may wonder how the different implementations can do it efficiently, here is the answer:

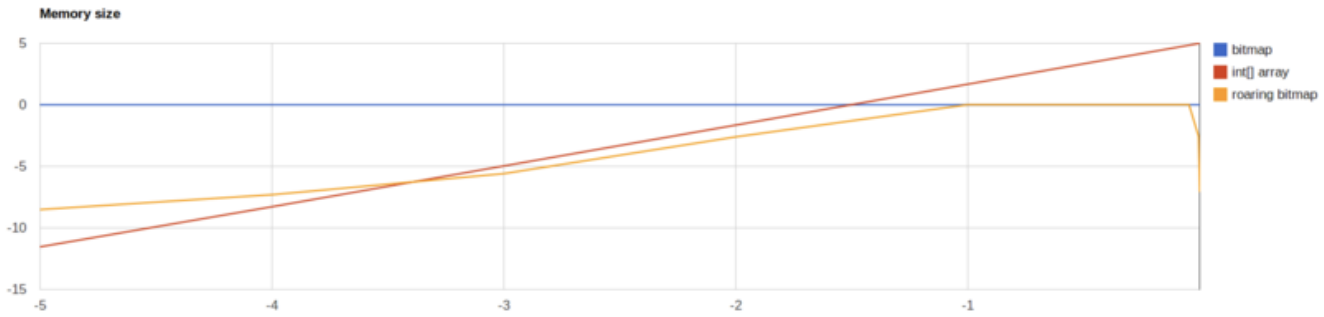
- “for" encodes **skip lists** on disk
- bitmaps rely on **finding the first bit which is set**
- roaring bitmaps are naturally indexed, we first go to the block that contains the target doc ID, and then either perform a binary search if documents are stored in an array or use the same approach as the bitmap otherwise
- the int[] array performs an **exponential search**

Even though the bitmap is still much slower than other implementations on sparse sets, it is also the fastest implementation on dense sets. The performance of other implementations compared to bitmaps degrades when density increases with roaring

bitmaps having the most graceful performance degradation.

You might wonder what explains the little jump that we can observe on very high densities on roaring bitmaps. The explanation is that our roaring bitmap implementation inverses its encoding when the set becomes very dense: instead of storing doc IDs which are contained in the set, it stores those that are missing. This makes skipping a bit slower but also helps with memory usage and can typically be useful for filter caching if you have some filters of the form “all except X".

Memory footprint



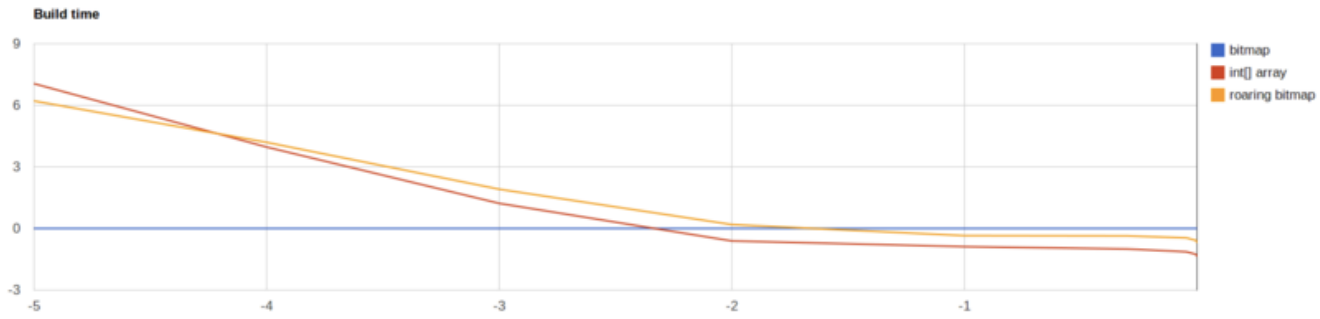
The less memory a cached filter uses, the more filters we will be able to cache, which makes good compression appealing. Here we can see that roaring bitmaps are almost always better than both the int[] array and a bitmap. The only exception is the very sparse case (less than 0.05% of documents contained in the set) where the memory overhead of each block makes roaring bitmaps less efficient than simple arrays.

Also the int[] array performs poorly on dense sets by requiring about 32x more memory than a bitmap or a roaring bitmap.

For this benchmark, we used a uniform distribution of documents. While this does not matter for the int[] and bitmap encodings, this is actually the worst-case for roaring bitmaps, so we could expect them to perform even better on more realistic data.

Note: “for" is absent from this benchmark as it is fully disk-based and does not require any memory at all.

Build time



One final yet important factor to consider for filter caching is the amount of time it takes to build a cache entry. And again roaring bitmaps are appealing by being either the fastest implementation (when density is between 0.01% and 1%) or very close to the fastest implementation (int[] array when the density is < 0.01%, and bitmap when density is > 1%).

Conclusion

There is no particular implementation which is constantly better than all others. However some implementations were disqualified because they performed poorly in some particular cases:

- bitmaps perform poorly on sparse sets, both in terms of performance and memory usage
- simple int[] arrays are fast but have crazy high memory usage on dense sets

Even though roaring bitmaps are rarely the fastest implementation, they are never a bad choice.

Another important conclusion from this comparison is that even though postings lists from the inverted index are stored on disk instead of memory, they remain very fast. The nextDoc benchmark (and to some extent the advance one too) showed that they are even competitive with an in-memory implementation. This means that the filter cache should only be used on slow filters in order to be efficient, and probably not on fast filters such as term filters.

Finally, Lucene today always uses the same implementation for all filters. We might make it more efficient in the future, for instance by using a different implementation depending on the density of the doc id set that we have to cache (patches welcome!).

Elastic Stack

Elasticsearch

Kibana

Beats

Logstash

订阅

定价

公司

招贤纳士

董事会

联系我们

中国区合作伙伴

资源

文档

ELK Stack 简介

Elasticsearch 简介

从 Splunk 迁移

AWS Elasticsearch 对比

关注我们

语言

简体中文