Blog
News (/blog/category/news)
Engineering (/blog/category/engineering)
User Stories (/blog/category/user-stories)
Releases (/blog/category/releases)
Culture (/blog/category/culture)
Archive (/blog/archive)

22 AUGUST 2013 **ENGINEERING (/BLOG/CATEGORY/ENGINEERING)**

You Complete Me

By **Alexander Reelsen (/blog/author/alexander-reelsen)**

Share

Effective search is not just about returning relevant results when a user types in a search phrase, it's also about helping your user to choose the best search phrases. Elasticsearch already has ***did-you-mean* (/guide/en/elasticsearch/reference/current/search-suggesters-phrase.html)** functionality which can correct the user's spelling after they have searched. Now, we are adding the ***completion suggester* (/guide/en/elasticsearch/reference/current/search-suggesters-completion.html)** which can make suggestions **while-you-type**. Giving the user the right search phrase before they have issued their first search makes for happier users and reduced load on your servers.

NOTE: Consider this feature experimental at the moment! Things might change/break in future releases.

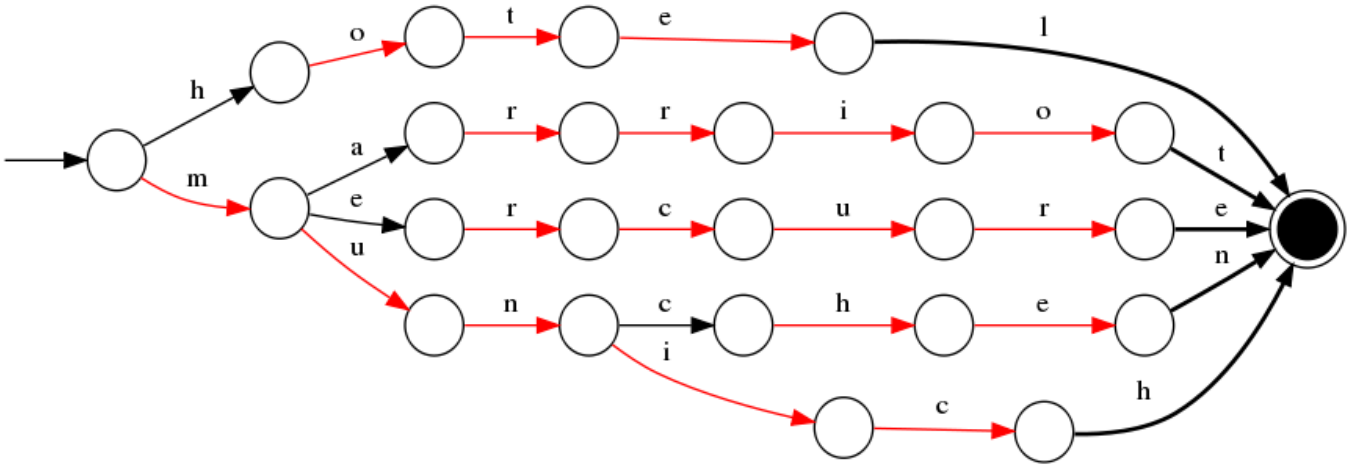
Why Another Suggester?

It was already possible to make suggestions using existing functionality in Elasticsearch, like prefix queries and ngrams, so why have we added a dedicated completion suggester? There are a few reasons:

SPEED

Remember, we're making suggestions while the user types, so results need to be shown to the user within a few milliseconds, even after taking network latency into account! A full-blown search has to examine too many terms (and their frequencies) to perform sufficiently fast for this purpose.

Instead, we use an in-memory data structure called an FST which contains valid suggestions and is optimised for fast retrieval and memory usage. Essentially, it is just a graph. For instance, and FST containing the words `hotel`, `marriot`, `mercure`, `munchen` and `munich` would look like this:



All we do is start on the left and follow the paths to the right. If the user types an `h`, then we can see that there is only one possible completion: `hotel`, so we can immediately complete that word. If the user types an `m`, then we can provide a list of all the "m" words. As soon as they type `ma`, then we can autocomplete the word "marriot". Following this in-memory graph is blazingly fast, as you will see from the benchmarks later in this blogpost.

Real Time

Suggesters in Lucene are built in-memory by loading the completion values from the index, then building the FST. This can be a slow, resource intensive process. And, as soon as the index changes, the FST needs to be rebuilt. "Real time search" is a mantra of Elasticsearch. It is not

acceptable to return out of date suggestions, nor to require a full rebuild whenever the index changes.

Instead of building the FST at search time, we now build an FST per-segment at index time. Essentially, whenever a new segment is written to disk, we also write the FST to a file in a format which is fast to load into memory when required. Instead of consulting a single FST for the whole index, we query each per-segment FST to produce a unified list of results.

Having an FST per segment also means that the completion suggester **scales horizontally**, in exactly the same way as for full text search: the more nodes you add, the more you can scale.

Readability

A user searching for `"Courtyard by Marriot, Munich City"` may type in any number of search phrases:

- courtyard munich
- munich hotel
- marriot munchen
- etc

However, we want our suggestions to be explicit, leaving the user in no doubt about the page they will see if they click on our suggestion. All of the above inputs should return the single, nicely formatted suggestion of `"Courtyard by Marriot, Munich City"`.

Custom Ordering

We want suggestions to be presented in a particular order, but this order is not the same as the ordering we need for full text search. Full text search uses TF/IDF (term frequency/inverse document frequency) to find the documents that are most relevant for the given search phrase. For suggestions, we may want common terms to appear at the top of the list. However common terms have high document frequencies, which **reduce** their relevance.

Alternatively, we may want a completely custom ordering applied to suggestions, which is independent of term frequency. For instance, we may want to promote hotels which have received good user ratings, or promote recent blogposts over older blogposts. The completion

suggester gives us complete control over the order of suggestions but, by default, treats more common suggestions as more important.

Use Case: Hotel Bookings

To demonstrate the power of the **completion suggester (/guide/en/elasticsearch/reference/current/search-suggesters-completion.html)**, let's start with a simple *search-as-you-type* function on a hotel bookings website. We'll build on this example below.

First, create an index, and setup the completion suggester for the `name_suggest` field:

```
curl -X PUT localhost:9200/hotels -d '
{
  "mappings": {
    "hotel" : {
      "properties" : {
        "name" : { "type" : "string" },
        "city" : { "type" : "string" },
        "name_suggest" : {
          "type" :     "completion"
        }
      }
    }
  }
}'
```

Note the new suggester `type` : `"completion"` . Then, index some hotels:

```
curl -X PUT localhost:9200/hotels/hotel/1 -d '
{
  "name" :         "Mercure Hotel Munich",
  "city" :         "Munich",
  "name_suggest" : "Mercure Hotel Munich"
}'
curl -X PUT localhost:9200/hotels/hotel/2 -d '
{
  "name" :         "Hotel Monaco",
  "city" :         "Munich",
  "name_suggest" : "Hotel Monaco"
}'
curl -X PUT localhost:9200/hotels/hotel/3 -d '
{
  "name" :         "Courtyard by Marriot Munich City",
  "city" :         "Munich",
  "name_suggest" : "Courtyard by Marriot Munich City"
}'
```

And now we can ask for suggestions:

```
curl -X POST localhost:9200/hotels/_suggest -d '
{
  "hotels" : {
    "text" : "m",
    "completion" : {
      "field" : "name_suggest"
    }
  }
}'
```

The (partial) response, as shown below, contains a single suggestion: `Mercure Hotel Munich` :

```
...
"hotels" : [
  {
    "text" : "m",
    "offset" : 0,
    "length" : 1,
    "options" : [
      {
        "text" : "Mercure Hotel Munich",
        "score" : 1.0
      }
    ]
  }
]
```

Why has it returned just `Mercure Hotel Munich`? Why doesn't it include `Hotel Monaco` or `Marriot`? The reason is that an FST query is not the same as a full text query. We can't find words anywhere within a phrase. Instead, we have to start at the left of the graph and move towards the right. The only suggestion that starts with an `m` is the `Mercure Hotel Munich`.

The solution to this is to provide multiple inputs: several possible phrases that the user may type.

Multiple Inputs

We'll reindex all of our hotels, providing multiple search phrases for each suggestion:

```
curl -X PUT localhost:9200/hotels/hotel/1 -d '
{
  "name" :         "Mercure Hotel Munich",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Mercure Hotel Munich",
      "Mercure Munich"
    ]
  }
}'
curl -X PUT localhost:9200/hotels/hotel/2 -d '
{
  "name" :         "Hotel Monaco",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Monaco Munich",
      "Hotel Monaco"
    ]
  }
}'
curl -X PUT localhost:9200/hotels/hotel/3 -d '
{
  "name" :         "Courtyard by Marriot Munich City",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Courtyard by Marriot Munich City",
      "Marriot Munich City"
    ]
  }
}'
```

Read Less ↑

If we rerun our completion suggester, we get more results:

```
"hotels" : [
  {
    "text" : "m",
    "offset" : 0,
    "length" : 1,
    "options" : [
      {
        "text" : "Marriot Munich City",
        "score" : 1.0
      },
      {
        "text" : "Mercure Hotel Munich",
        "score" : 1.0
      },
      {
        "text" : "Mercure Munich",
        "score" : 1.0
      },
      {
        "text" : "Monaco Munich",
        "score" : 1.0
      }
    ]
} ]
```

Great! We're now seeing all the hotels which start with `m`. But it's not quite perfect: the `mercure hotel` has been returned twice; once as `Mercure Hotel Munich` and once as `Mercure Munich`. This is because both inputs matched the search text. We want to unify those results into a single, standardised output.

Single Unified Output

We'll reindex our documents, specifying the text of the suggestion that should be returned whenever one of the `input`s is matched:

```
curl -X PUT localhost:9200/hotels/hotel/1 -d '
{
  "name" :         "Mercure Hotel Munich",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Mercure Hotel Munich",
      "Mercure Munich"
    ],
    "output":      "Hotel Mercure"
  }
}'
curl -X PUT localhost:9200/hotels/hotel/2 -d '
{
  "name" :         "Hotel Monaco",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Monaco Munich",
      "Hotel Monaco"
    ],
    "output":      "Hotel Monaco"
  }
}'
curl -X PUT localhost:9200/hotels/hotel/3 -d '
{
  "name" :         "Courtyard by Marriot Munich City",
  "city
```

Read More ↓

Finally, the results from our suggester are correct: we have a single result for each matching document, and the suggestions are formatted as we want them:

```
"hotels" : [
  {
    "text" : "m",
    "offset" : 0,
    "length" : 1,
    "options" : [
      {
        "text" : "Hotel Marriot",
        "score" : 1.0
      },
      {
        "text" : "Hotel Mercure",
        "score" : 1.0
      },
      {
        "text" : "Hotel Monaco",
        "score" : 1.0
      }
    ]
} ]
```

Scoring - The End of Neutrality

Not all suggestions have equal value. Perhaps our users have ranked some hotels higher than others. Or perhaps we earn more revenue from some hotels than from others. Either way, we want to be able to control the order in which suggestions are returned.

We'll reindex our documents, this time specifying a `weight` for each hotel:

```
curl -X PUT localhost:9200/hotels/hotel/1 -d '
{
  "name" :         "Mercure Hotel Munich",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Mercure Hotel Munich",
      "Mercure Munich"
    ],
    "output":      "Hotel Mercure",
    "weight":      5
  }
}'
curl -X PUT localhost:9200/hotels/hotel/2 -d '
{
  "name" :         "Hotel Monaco",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Monaco Munich",
      "Hotel Monaco"
    ],
    "output":      "Hotel Monaco",
    "weight":      10
  }
}'
curl -X PUT localhost:9200/hotels/hotel/3 -d '
{
  "name
```

Read More ↓

As you can imagine, the results are now sorted by `weight`, which is returned as the `score`:

```
"hotels" : [
  {
    "text" : "m",
    "offset" : 0,
    "length" : 1,
    "options" : [
      {
        "text" : "Hotel Marriot",
        "score" : 15.0
      },
      {
        "text" : "Hotel Monaco",
        "score" : 10.0
      },
      {
        "text" : "Hotel Mercure",
        "score" : 5.0
      }
    ]
} ]
```

Note: A weight must be an integer (not a float, as you are used from normal scoring) between 0 and 2^31.
Note: If you don't specify a `weight` then Elasticsearch will use the term frequency of the search phrase within its segment, usually `1`. This is pretty much meaningless as far as suggestions go. It is better to control order using `weight`.

Connecting Users With Results

The question arises: what does the user do with our suggestions? We want to connect the user with the correct results as fast as possible. If we are returning explicit suggestions, eg `Courtyard by Marriot, Munich City`, then why should we force the user to perform a search on those words? Why not take the user directly to the correct page?

We can enrich the suggestions by including a `payload`, which can be an arbitrary JSON document. For our hotel suggestions, we can include a simple JSON document which includes the ID of the hotel. Payloads must be enabled in the mapping before we can use them, which means that we need to recreate the index:

```
curl -X DELETE localhost:9200/hotels
curl -X PUT localhost:9200/hotels -d '
{
  "mappings": {
    "hotel" : {
      "properties" : {
        "name" : { "type" : "string" },
        "city" : { "type" : "string" },
        "name_suggest" : {
          "type" :     "completion",
          "payloads" : true
        }
      }
    }
  }
}'
```

And reindex the hotels with the appropriate payloads:

```
curl -X PUT localhost:9200/hotels/hotel/3 -d '
{
  "name" :         "Courtyard by Marriot Munich City",
  "city" :         "Munich",
  "name_suggest" : {
    "input" :      [
      "Courtyard by Marriot Munich City",
      "Marriot Munich City"
    ],
    "output":      "Hotel Marriot",
    "weight":      15,
    "payload":     { "hotel_id": 3}
  }
}'
```

The suggester responses will return whatever `payload` was associated with that suggestion:

```
...
{
  "text" :    "Hotel Marriot",
  "score" :   15.0,
  "payload" : { "hotel_id": "3" }
},
...
```

When the user clicks on `Hotel Marriot`, we can transfer them directly to the correct page, without having to fire off another search request.

Working With Synonyms

The completion suggester can make use of synonyms, in the same way as normal searches. For example, let's make `courtyard` and `marriot` synonoms of each other:

```
curl -X DELETE localhost:9200/hotels
curl -X PUT localhost:9200/hotels -d '
{
  "settings": {
    "analysis": {
      "analyzer": {
        "suggest_synonyms": {
          "type":      "custom",
          "tokenizer": "lowercase",
          "filter":    [ "my_synonyms" ]
        }
      },
      "filter": {
        "my_synonyms": {
          "type":     "synonym",
          "synonyms": [ "courtyard, marriot" ]
        }
      }
    }
  },
  "mappings": {
    "hotel" : {
      "properties" : {
        "name" : { "type" : "string" },
        "city" : { "type" : "string" },
        "name_suggest" : {
          "type" :            "completion
```

Read More ↓

Now if we index the `Courtyard Hotel`, it will be added to the FST as `(courtyard|marriot)` `hotel`:

```
curl -X PUT 'localhost:9200/hotels/hotel/3?refresh=true' -d '
{
  "name" :         "Courtyard Hotel",
  "city" :         "Munich",
  "name_suggest" : "Courtyard Hotel"
}'
```

Suggestions for the letter `m` will match on `marriot hotel` and return the `Courtyard Hotel` as a suggestion. Be aware that this might confuse the user, so you should add enough context to the output label so that the user understands why a suggestion has been included.

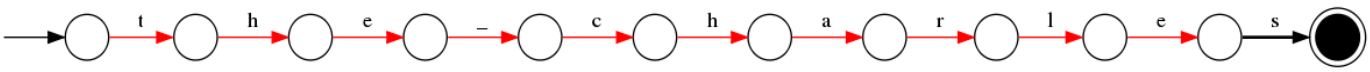
Note: In version 0.90.3, `index_analyzer` and `search_analyzer` have to be specified separately. From version 0.90.4 onwards, you will be able to use the single `analyzer` parameter instead.

By default, the **`simple analyzer` (http://www.elasticsearch.org/guide/reference/index-modules/analysis/simple-analyzer/)** is used at both index and search time.

Ignoring Stopwords

Stopwords also come into play for suggestions. A user searching for `The Charles Hotel` might well type in just `Charles Hotel`. We should still be able to find the right hotel. Of course, you could just index it with two `input`s: `["Charles Hotel", "The Charles Hotel"]`, but you could use stopwords to do it for you automatically.

Unfortunately, there is a bit of a gotcha with stopwords. The stopwords token filter removes terms, but still leaves a blank in the token stream, where the term used to be. For instance, without stopwords, `The Charles` would generate a graph like the following, where `_` represents the space between the two words:



With stopwords, however, the suggestion now STARTS with a space:

To get around this, we have to change the completion suggester mapping to set both `preserve_position_increments` and `preserve_separators` to `false`:

```
curl -X DELETE localhost:9200/hotels
curl -X PUT localhost:9200/hotels -d '
{
  "mappings": {
    "hotel" : {
      "properties" : {
        "name" : { "type" : "string" },
        "city" : { "type" : "string" },
        "name_suggest" : {
          "type" :            "completion",
          "index_analyzer" :  "standard",
          "search_analyzer" : "standard",
          "preserve_position_increments": false,
          "preserve_separators": false
        }
      }
    }
  }
}'
```

Now, `The Charles Hotel` is added to the FST without stopwords or separators, ie as `charleshotel`.

Note: This can have unintended consequences, as `charlesh` will now match `The Charles Hotel`. Similarly, searching for `simon t` will not match `Simon the Sorcerer` as `t` is not a stopword. While using stopwords can be a time saver, you will get better results by manually specifying all the ìnput variations that you want to be able to match.

Note: The added complication that stopwords brings to the suggester is also the reason why the default analyzer is the `simple` analyzer (which doesn't use stopwords) instead of the `standard` analyzer.

Easy Updates

While Elasticsearch makes it easy to add *ad hoc* suggestions to existing documents, it will often make more sense to maintain the suggestions in a separate index. Payloads allow you to link the suggestions back to the original resource.

Improving Relevance

Maintaining a good suggestions list requires for thought and regular maintenance. It takes mores effort than straightforward full text search, but that effort will be repaid by better user experience and increased conversion rates.

Fine tuning suggestions requires to combine several signals:

- Log your searches, find the most important ones, add good suggestions first
- Log the suggestions which were selected by the user (together with the searches)
- Refine your inputs based on the above steps
- Use weights with a reasonable logic behind them - one which adheres to your revenue model or whatever your definition of a best result is

Further Plans

After laying the foundations for this powerful completion framework, we will start adding new features gradually. Here are a few in the pipeline:

Going Fuzzy

In order to find suggestions even if the user misspells some words, the completion suggester will support fuzzy queries as of elasticsearch 0.90.4. All you have to do is to add the `fuzzy` option to your suggest request:

```
curl -X POST 'localhost:9200/hotels/_suggest?pretty' -d '
{
  "hotels" : {
    "text" : "coutr",
    "completion" : {
      "field" : "name_suggest",
      "fuzzy" : {
        "edit_distance" : 2
      }
    }
  }
}'
```

Improved Stopwords Support

Another issue when using stopwords with suggesters is that stopwords like `a` can interfere with autocompletion of words like `apple`. The standard stopwords filter will remove `a` so we are not able to provide suggestions until the user has typed a second letter.

Version 0.90.4 will support the new **SuggestStopFilter (https://issues.apache.org/jira/browse/LUCENE-5165)** which Mike McCandless built specifically for suggesters: if the stopword is the final word in the query and is not followed by a space or other word boundary, then it will not be treated as a stopword but rather as a prefix that can be queried.

Statistics

Having insight into how much RAM is consumed by the in-memory FST is important for capacity planning. Roughly similar to the fielddata statistics, you will be able to get total and per-field memory usage. This feature will also land in 0.90.4.

```
curl 'http://localhost:9200/_stats/completion?pretty&human'
{
  "ok" : true,
  "_shards" : {
    "total" : 10,
    "successful" : 5,
    "failed" : 0
  },
  "_all" : {
    "primaries" : {
      "completion" : {
        "size_in_bytes" : 163691700,
        "size": "156.1mb"
      }
    },
    "total" : {
      "completion" : {
      "size_in_bytes" : 163691700,
      "size": "156.1mb"
      }
    }
  },
  "indices" : {
    "data" : {
      "primaries" : {
        "completion" : {
          "size_in_bytes" : 163691700,
          "size": "156.1mb"
        }
      },
      "total" : {
        "completion" : {
          "size_in_bytes" : 163691700,
          "size
```

Read More ↓

Highlighting

Many *search-as-you-type* implementations highlight the portion of the term which has already been entered. We plan to add this to the response as we well.

Reduce Payload Memory Usage

As you will see from the benchmarks below, even small payloads can use a significant amount of memory. Version 0.90.4 will support payloads as simple values (ie integers, strings etc) rather than requiring a JSON body. Also, we will be looking at storing payloads on disk with block compression, to make their impact on memory usage negligible.

Performance & Benchmarks

In order to back the claims of incredible speed, I include these stats from a short benchmark on my developer notebook with an SSD and enough RAM to fit the whole index into the file system cache, which makes the regular `prefix` and `match_phrase_prefix` queries significantly faster than the typical case. Tests were performed over the loopback interface.

The dataset consisted of the titles of 2.1 million pages in Wikipedia.

	completion	fuzzy 1	fuzzy 2	prefix query	match query	FST size in-memory
standard	0.72ms	1.14ms	4.14ms	4.41ms	5.80ms	82mb
optimized	0.53ms	0.69ms	2.46ms	3.33ms	1.63ms	80mb
payload	0.65ms	1.16ms	4.32ms	4.22ms	6.18ms	166mb
payload + optimized	0.62ms	0.75ms	2.72ms	3.30ms	1.60ms	163mb

Explanation

- `completion` : a simple `completion suggest` request
- `fuzzy 1` : a `completion suggest` request with fuzzy matching, `edit_distance` of 1
- `fuzzy 2` : a `completion suggest` request with fuzzy matching, `edit_distance` of 2
- `match` : a `match_phrase_prefix` full text query
- `prefix` : a simple `prefix` full text query
- `optimized` : the index was optimized down to a single segment before running the tests

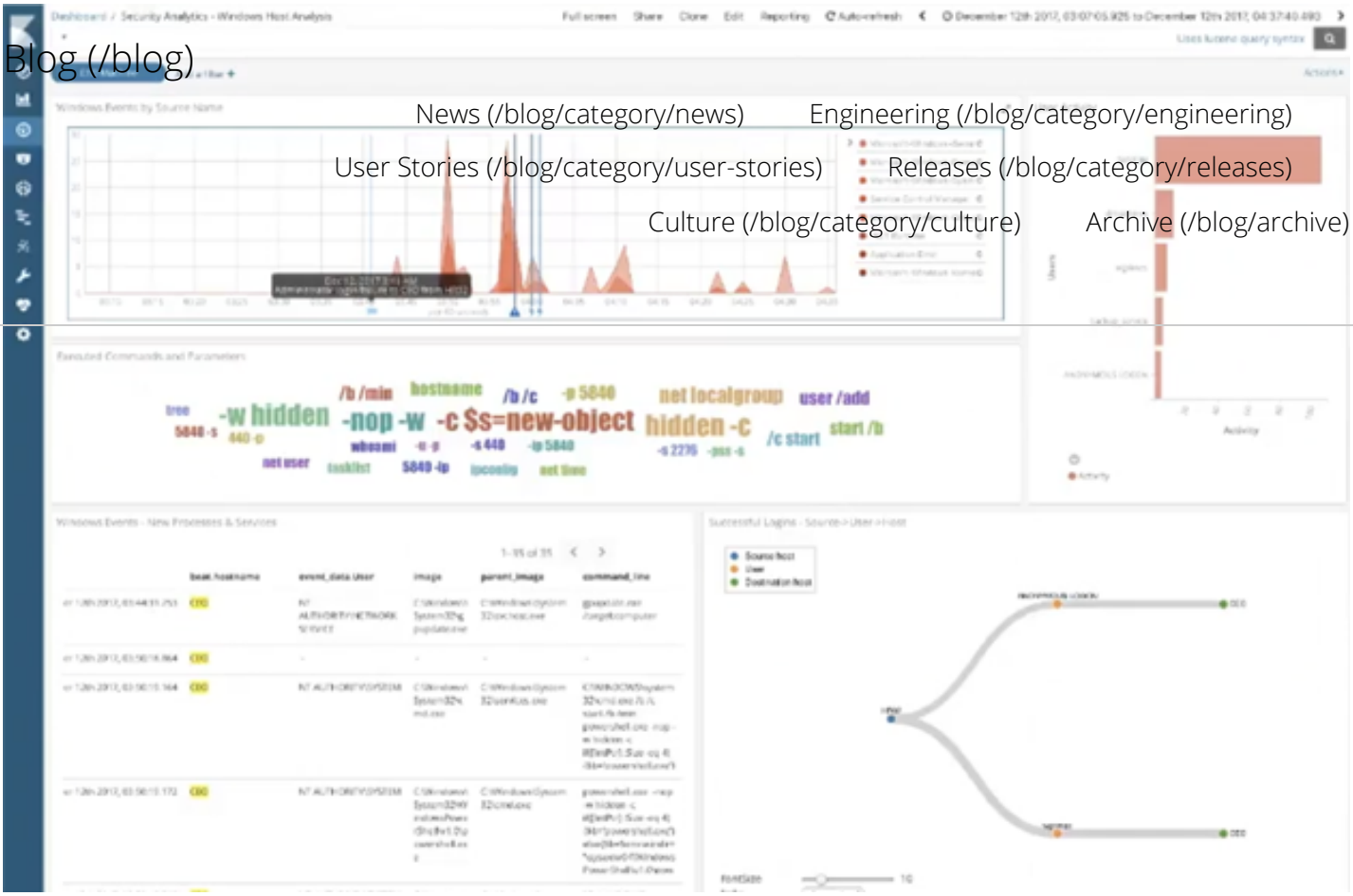
As you can see from the results, even the `fuzzy 2` request is blazingly fast.

Note: Adding the `payload` (which was small) doubled the amount of memory used by the FST. Try to keep the payload as small as possible.

Round-up

That's it for now. You have had an overview of the past, the present and the future of the completion suggester, and hopefully you are raring to start using it into your application. We will keep you posted, once we add some of the features planned. Also keep in mind, that this feature is still marked experimental.

If you have an interesting use case for this suggester you would like to share, please do so, we are happy to get some feedback about it.



(https://events.elastic.co/2018-12-07-cybersecuritywebinar-APAC/rtp?iesrc=ctr)

Learn more how the latest security related features in 6.x Elastic Stack can help you to solve your cybersecurity difficulties with our cyber security advocate, Kevin Keeney on 7 Dec 2018.

Register Now (https://events.elastic.co/2018-12-07-cybersecuritywebinar-APAC/rtp?iesrc=ctr)

Recommended Content

Elasticsearch: Getting Started (https://www.elastic.co/webinars/getting-...

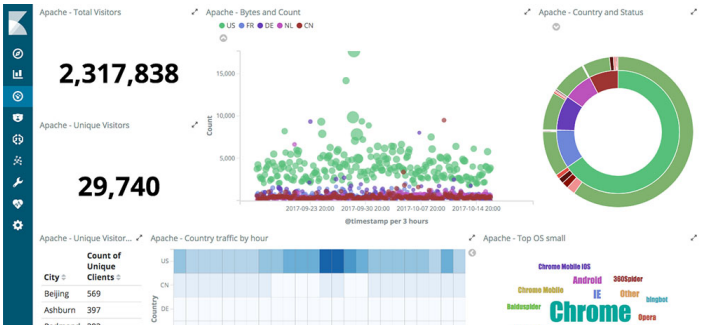
Get an understanding of what Elasticsearch is capable of, how to implement basic functionality, and where to find more

resources.

Beats and Marvel to Monitor Infrastructure (https://www.elastic.co/webinars/using-...

(Video 58:18)

Learn More (https://www.elastic.co/webinars/using-beats-and-marvel-to-monitor-your-infrastructure?baymax=rtp&storm=recommendation&elektra=blog&iesrc=rcmd&astid=c2ab6d53-b48f-4f42-9f76-b39be9f6b0b6&at=77&rcmd_source=WIDGET&req_id=47901c9b-b603-4bc5-b2e1-e6af416b8280)



Getting Started with Kibana (https://www.elastic.co/webinars/getting-...

This video is perfect for users that are new to Kibana and are looking for a primer on data exploration, visualization, and dashboarding.

Learn More (https://www.elastic.co/webinars/getting-started-kibana?baymax=rtp&storm=recommendation&elektra=email&iesrc=rcmd&astid=85e6b18c-960c-483d-9c76-9de65f440340&at=77&rcmd_source=WIDGET&req_id=47901c9b-b603-4bc5-b2e1-e6af416b8280)

Blog (/blog)

Documentation (/guide)

Elastic{ON} Events (/elasticon)

News (/blog/category/news) Engineering (/blog/category/engineering)

Forums (https://discuss.elastic.co/)

User Stories (/blog/category/user-stories) Releases (/blog/category/releases)

Meetups (/community/meetups)

Culture (/blog/category/culture) Archive (/blog/archive)

Subscriptions (/subscriptions)

Support Portal (https://support.elastic.co)

Videos & Webinars (/videos)

Training (/training)

ABOUT > (/ABOUT)

Careers/Jobs (/about/careers)

Our Source Code (/about/our-source-code)

Teams (/about/teams)

Board of Directors (/about/teams/board)

Leadership (/about/teams/leadership)

Contact (/contact)

Our Story (/about/history-of-elasticsearch)

Why Open Source (/about/why-open-source)

Distributed by Intention (/about/distributed)

Partners (/about/partners)

Media (/about/press)

Investor Relations (https://ir.elastic.co)

Elastic Store (https://www.elastic.shop)

()

FOLLOW US

Trademarks (/legal/trademarks) / Terms of Use (/legal/terms-of-use) / Privacy (/legal/privacy-statement) / Brand (/brand)

elastic

(/)

Blog (/blog)

News (/blog/category/news) Engineering (/blog/category/engineering)

User Stories (/blog/category/user-stories) Releases (/blog/category/releases)

Culture (/blog/category/culture) Archive (/blog/archive)