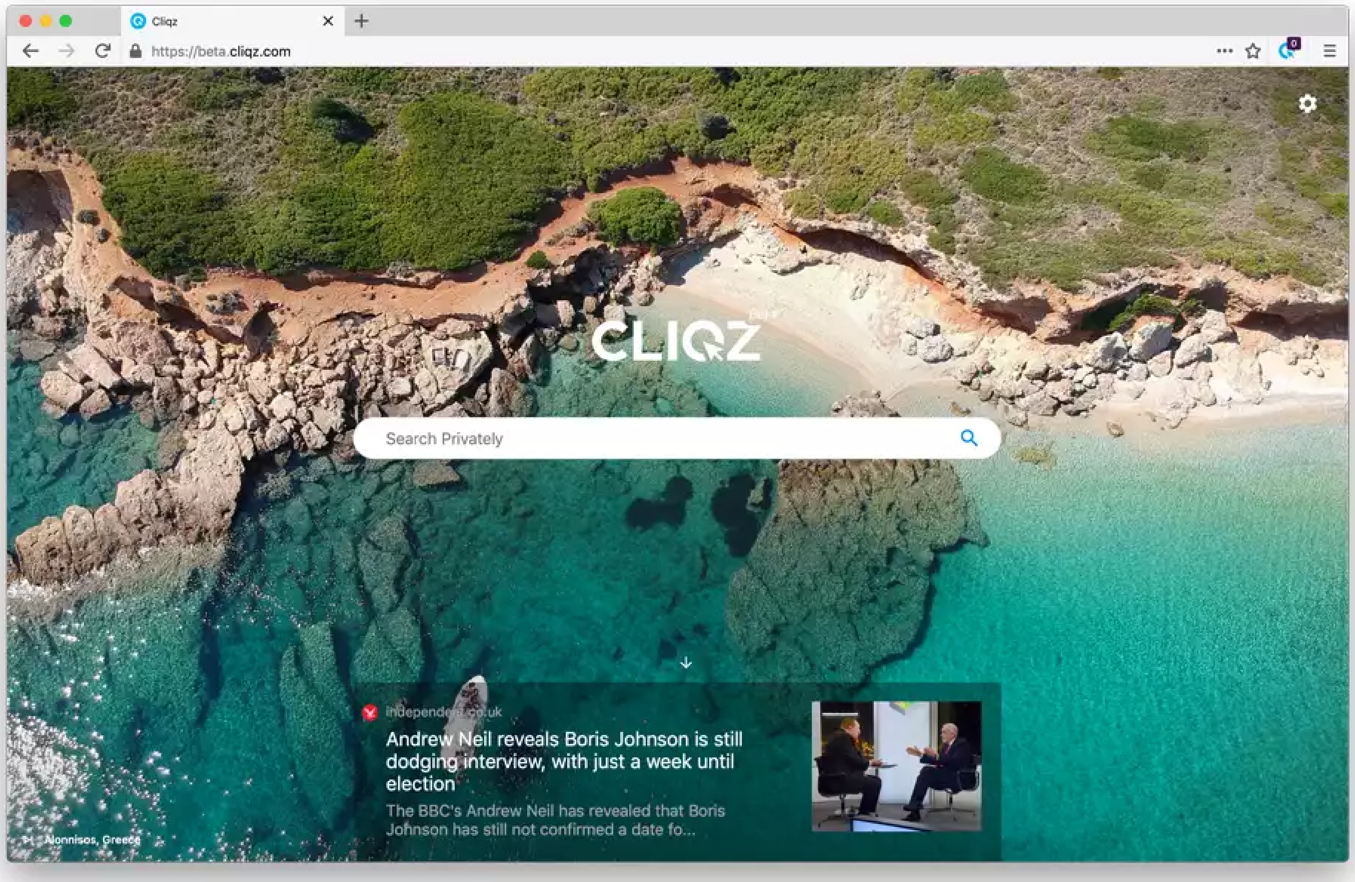
Tech @ Cliqz

privacy search web

Building a search engine from scratch

A whirlwind tour of the big ideas powering our web search


Cliqz
https://beta.cliqz.com
CLIQZ
Search Privately
independent.co.uk
Andrew Neil reveals Boris Johnson is still dodging interview, with just a week until election
The BBC's Andrew Neil has revealed that Boris Johnson has still not confirmed a date fo...
Alonnisos, Greece

December 6th, 2019

The previous blog post in this series explored our journey so far in building an independent, alternative search engine. If you haven't read it yet, we would highly recommend checking it out first!

It is no secret that Google search is one of the most lucrative businesses on the planet. With quarterly revenues of Alphabet Inc. exceeding $40 Billion[1] and a big portion of that driven by the advertising revenue on Google's search properties, it might be a little surprising to see the lack of competition to Google in this area[2]. We at Cliqz believe that this is partly due to the web search *bootstrapping* problem: the entry barriers in this field are so massive that the biggest, most successful companies in the world with the resources to tackle the problem shy away from it. This post attempts to detail the bootstrapping problem and explain the Cliqz approach to overcoming it. But let us first start by defining the search problem.

A toy search engine

The expectation for a modern web search engine is to be able to answer any user question with the most relevant documents that exist for the topic on the internet. The search engine is also expected to be blazingly fast, but we can ignore that for the time being. At the risk of gross oversimplification, we can define the web search task as computing the **content match** of each candidate document with respect to the user question (**query**), computing the current **popularity** of the document and combining these scores with some heuristic.

The content match score measures how well a given document matches a given query. This could be as simple as an exact keyword match, where the score is proportional to the number of query words present in the document:

query	avengers endgame
document	avengers endgame imdb

If we could score all our documents this way, filter the ones that contain all the query words and sort them based on some *popularity* measure, we would have a functioning, albeit toy, search engine. Let us look at the challenges involved in building a system capable of handling *just* the exact keyword match scenario at a web scale, which is a **bare minimum** requirement of a modern search engine.

The Bootstrapping Problem

According to a study published on worldwidewebsize.com, a conservative estimate of the number of documents indexed by Google is around 60 Billion.

1. The infrastructure costs involved in serving a massive, constantly updating inverted index at scale.

Considering just the text content of these documents, this represents at least a petabyte of data. A linear scan through these documents is technically not feasible, so a well understood solution to this problem is to build an *inverted index*. The big cloud providers like Amazon, Google or Microsoft are able to provide us with the infrastructure needed to serve this system, but it is still going to cost millions of euros each year to operate. And remember, this is just to **get started**.

2. The engineering costs involved in crawling and sanitizing the web at scale.

The *crawler*[3] infrastructure needed to keep this data up to date while detecting newer documents on the internet is another massive hurdle. The crawler needs to be *polite* (some form of domain level rate-limiting), be geographically distributed, handle multilingual data and aggressively avoid link farms and spider traps[4]. A huge portion of the crawlable[5] web is spam and duplicate content; sanitizing this data is another big engineering effort.

Also, a significant portion of the web is cut off from you if your crawler is not *famous*. Google has a huge competitive advantage in this regard, a lot of site owners allow just *GoogleBot* (and maybe *BingBot*), making it an extremely tedious process for an unknown crawler to get whitelisted on these sites. We would have to handle these on a case-by-case basis knowing that getting the attention of the sites is not guaranteed.

3. You need users to get more users (Catch-22)

Even assuming we manage to index and serve these pages, measuring and improving the *search relevance* is a challenge. Manual evaluation of search results can help get us started, but we would need real users to measure changes in search relevance in order to be competitive.

4. Finding the relevant pages amongst all the noise on the web.

The biggest challenge in search, though, is the **removal of noise**. The Web is so vast that any query can be answered. The ability to discard the noise in the process makes all the difference between a useless search engine and a great one. We discussed this topic with some rigor in our previous post, providing a rationale for why using query logs is a smarter way to cut through the noise on the web. We also wrote in depth about how to collect these logs in a responsible manner using Human Web. Feel free to check these posts out for further information.

Cliqz Search 101: Query Logs, Page Models

Query/URL pairs, typically referred to as query logs, are often used by search engines to optimize their ranking and for SEO to optimize incoming traffic. Here is a sample from the AOL query logs dataset[6].

Query	Clicked Url
google	http://www.google.com
wnmu	http://www.wnmu.edu
ww.vibe.com	http://www.vibe985.com
www.accuweather.com	http://www.accuweather.com
weather	http://asp.usatoday.com
college savings plan	http://www.collegesavings.org
pennsylvania college savings plan	http://www.patreasury.org

Query	Clicked Url
pennsylvania college savings plan	http://swz.salary.com

We can use these query logs to build a model of the page *outside* of its content, which we refer to as **page models**. The example below comes from a truncated version of the page model that we have at the moment for one particular CNN article on Tesla's Cybertruck launch. The scores associated with the query are computed as a function of its *frequency* (i.e. the number of times the query/URL pair was seen in our logs) and its *recency* (i.e. recently generated query logs are a better predictor for relevance).

```json
{
  "queries": [
    [
      "tesla cybertruck",
      0.5111737168808949
    ],
    [
      "tesla truck",
      0.4108341455983614
    ],
    [
      "new tesla car",
      0.022784473090294764
    ],
    ...
    ...
    ...
    [
      "pick up tesla",
      0.020538972510725183
    ],
    [
      "new tesla truck",
      0.019462471432017632
    ],
    [
      "cyber truck tesla price",
      0.006587470155023614
    ],
    [
      "how much is the cybertruck",
      0.003764268660013494
    ],
    ...
    ...
    ...
    [
      "cybertruck unveiling",
      0.0016181605575564585
    ],
    [
      "new tesla cybertruck",
      0.0016181605575564585
    ],
    [
      "cyber truck unveiling",
      0.0016181605575564585
    ]
  ]
}
```

We have hundreds of queries on the page, but even this small sample should provide you with an intuition on how a page model helps us summarize and understand the contents of the page. Even without the actual page text, the page model suggests that the article is about a new Tesla car called the *Cybertruck*; it details an unveiling event and it contains potential pricing information.

The more unique queries we can gather for a page, the better our model for the page will be. Our use of Human Web also enables us to collect anonymous statistics on the page, a part of which is shown below. This structure shows the popularity of the page in different countries at this moment in time, which is used as a popularity signal. We can see that it is very popular in the UK, less so is Australia, etc.

```json
    "counters": {
      "ar": 0.003380009657170449,
      "at": 0.016900048285852245,
      "au": 0.11492032834379527,
      "be": 0.02704007725736359,
      ...
      ...
      ...
      "br": 0.012071463061323033,
      "cn": 0.0014485755673587638,
      "cz": 0.008691453404152583,
      "de": 0.06422018348623854,
      "dk": 0.028971511347175277,
      "fr": 0.025108643167551906,
      ...
      ...
      ...
      "gb": 0.3355866731047803,
      "it": 0.00772573635924674,
      "jp": 0.005311443746982134,
      "ru": 0.0159343312409464,
      "se": 0.0294543698696282,
      "ua": 0.012071463061323033
      ...
    }
```

The Search Process

Now that we understand how page models are generated, we can start stepping through the search process. We can break down this process into different stages, as described below.

The TL;DR Version

This is a high level overview if you want to know how Cliqz search is different.

1. Our model of a web page is based on **queries** only. These queries could either be *observed* in the query logs or could be *synthetic*, i.e. we generate them. In other words, during the recall phase, we **do not** try to match query words directly with the content of the page. This is a crucial *differentiating* factor – it is the *reason* we are able to build a search engine with dramatically less resources in comparison to our competitors.
2. Given a query, we first look for *similar* queries using a multitude of keyword and word vector based matching techniques.

> 3. We pick the most similar queries and fetch the *pages* associated with them.
> 4. At this point, we start considering the *content* of the page. We utilize it for feature extraction during ranking, filtering and dynamic snippet generation.

1. The Query Correction Stage

When the user enters a query into our search box, we first perform some potential corrections on it. This not only involves some normalization, but also *expansions* and *spell corrections*, if necessary. This is handled by a service which is called *suggest* – we will have a post detailing its inner workings soon. For now, we can assume that the service provides us with a list of possible alternatives to the user query.

2. The Recall and Precision Stages

We can now start building the core of the search engine. Our index contains *billions* of pages; the job of the search is to find the *N*, usually around 50, most relevant pages for a given query. We can broadly split this problem into 2 parts, the *recall* and the *precision* stages.

The recall stage involves narrowing down the billions of pages to a much smaller set of, say, five hundred *candidate* pages, while trying to get as many relevant pages as possible into this set. The precision stage performs more intensive checks on these candidate pages to filter the top *N* pages and decide on the final ordering.

2.1 Recall Stage

A common technique used to perform efficient retrieval in search is to build an inverted index. Rather than building one with the words on the page as *keys*, we use ngrams of the queries on the page model as keys. This let's us build a much smaller and less noisy index.

We can perform various types of matching on this index:

- **Word based exact match**: We look for the exact user query in our index and retrieve the linked pages.

- **Word based partial match**: We split the user query into *ngrams* and retrieve the linked pages for each of these.
- **Synonym and stemming based partial match**: We *stem* the words in the user query and retrieve the linked pages for each of its ngrams. We could also replace words in the query with their synonyms and repeat the operation. It should be clear that this approach, if not used with caution, could quickly result in an explosion of candidate pages.

This approach works great for a lot of cases, e.g. when we want to match model numbers, names, codes, *rare-words*. Basically when the token of the query is a must, which is not possible to know before-hand. But it can also introduce a lot of noise as shown below, because we lack a semantic understanding of the query.

query	soldier of fortune game
document 1	ps2 game soldier fortune
document 2	soldier of fortune games
document 3 (Noise)	fortune games
document 4 (Noise)	soldier games

An alternative approach to recall is to map these queries to a vector space and match in this higher dimensional space. Each query gets represented by a K dimensional vector, 200 in our case, and we find the nearest neighbors to the user query in this vector space.

This approach has an advantage with the fact that it can match semantically. But it could also introduce noise with aggressive semantic matching, as illustrated below. This technique could also be unreliable when the query contains rare words, like model numbers or names, as their vector space neighbors could be random.

query	soldier of fortune game

document 1	soldier of fortune android
document 2	sof play
document 3	soldier of fortune playstation
document 4 (Noise)	defence of wealth game

We train these query vectors based on some custom word embeddings learnt from our data. These are trained on billions of <**good query**, **bad query**> pairs collected from our data and use the *Byte-Pair-Encoding* implementation of SentencePiece[7] to address the issue with missing words in our vocabulary.

We then build an index with billions of these query vectors using Granne[8], our in-house solution for memory efficient vector matching. We will have a post on the internals of Granne soon on this blog, you should definitely look out for it if the topic is of interest.

We also generate **synthetic** queries out of *titles*, *descriptions*, *words* in the URL and the actual *content* of the page. These are, by definition, noisier that the query-logs captured by HumanWeb. But they are needed; otherwise, newer pages with fresh content would not be retrievable.

Which model do we prefer? The need for semantics is highly dependent on the context of the query which is unfortunately difficult to know *a priori*. Consider this example:

Scenario 1	how tall are people in stockholm?	how tall are people in sweden?
Scenario 2	restaurants in stockholm	restaurants in sweden

As you can see, the *same* semantic matching could result in good or bad matches based on the query. The semantic matching in *Scenario 1* is useful for us to retrieve good results, but the matching in *Scenario* 2 could provide irrelevant results. Both keyword and vector based models have their strength and weaknesses. Combining them in an ensemble, together with multiple variations of those models, gives us much better results than any particular model in isolation. As one would expect, there is no *silver bullet model* or *algorithm* that does the trick.

The recall stage combines the results from both the keyword and vector-based indices. It then scores them with some basic heuristics to narrow down our set of candidate pages. Given the strict latency requirements, the recall stage is designed to be as quick as possible while providing acceptable recall numbers.

2.2 Precision Stage

The top pages from the recall stage enter the precision stage. Now that we are dealing with a smaller set of pages, we can subject them to additional scrutiny. Though the earlier versions of Cliqz search used a heuristic driven approach for this, we now use gradient boosted decision trees[9] trained on hundreds of heuristic and machine learned features. These are extracted from the query itself, the content of the page and the features provided by Human Web. The trees are trained on manually rated search results from our search quality team.

3. Filter Stage

The pages that survive the precision stage are now subject to more checks, which we call *filters*. Some of these are:

- **Deduplication Filter**: This filter improves diversity of results by removing pages that have duplicated or similar content.
- **Language Filter**: This filter removes pages which do not match the user's language or our acceptable languages list.
- **Adult Filter**: This filter is used to control the visibility of pages with adult content.
- **Platform Filter**: This filter replaces links with platform appropriate ones e.g. mobile users would see the mobile version of the web page, if available.
- **Status Code Filter**: This filter removes obsolete pages, i.e. links we cannot resolve anymore.

4. Snippet Generation Stage

Once the result set is finalized, we enhance the links to provide well formatted and *query-sensitive* snippets. We also extract any relevant structured data the page may contain to enrich its snippet.

The final result set is returned back to the user through our API gateway to the various frontend clients.

Other Important Topics Of Consideration

Maintaining Multiple Mutable Indices

The section on recall presented an extremely basic version of the index for the sake of clarity. But in reality, we have multiple versions of the index running in various configurations using different technologies. We use Keyvi[10], Granne, RocksDB and Cassandra in production to store different parts of our index based on their mutability, latency and compression constraints.

The total size of our index currently is around **50 TB**. If you could find a server with the required disk space along with sufficient RAM, it is possible to run our search on **localhost**, completely disconnected from the internet. It doesn't get more *independent* than that.

Search Quality

Search quality measurement plays an integral part in how we test and build our search. Apart from automated sanity checking of results against our competitors, we also have a dedicated team working on manually rating our results. Over the years, we have collected the ratings for millions of results. These ratings are used not only to test the system but to help train the ranking algorithms we mentioned before.

Fetcher

The query logs we collect from Human Web are unfortunately insufficient to build a good quality search. The actual content of the page is not only necessary to perform better ranking, it is required to be able to provide a richer user experience. Enriching the result with *titles*, *snippets*, *geolocation* and *images* helps the user make an informed decision about visiting a page.

It may seem like Common Crawl would suffice for this purpose, but it has poor coverage outside of the US and its update frequency is not realistic for use in a search engine.

While we do not *crawl* the web in the traditional sense, we maintain a distributed fetching infrastructure spread across multiple countries. Apart from fetching the pages in our index at periodic intervals, it is designed to respect politeness constraints and *robots.txt* while also dealing with blocklists and crawler unfriendly websites.

We still have issues getting our fetcher **cliqzbot** whitelisted on some very popular domains, like Facebook, Instagram, LinkedIn, GitHub and Bloomberg. **If you can help in any way**, please do reach out at beta[at]cliqz[dot]com. You'd be improving our search a lot!

Tech Stack

- We maintain a hybrid deployment of services implemented primarily in Python, Rust, C++ and Golang.
- Keyvi is our main index store. We built it to be space efficient, fast but also provide various approximate matching capabilities with its FST data structure.
- The mutable part of our indices are maintained on Cassandra and RocksDB.
- We built Granne to handle our vector based matching needs. It is a lot more memory efficient than other solutions we could find – you can read more about it in tommorrow's blog post.
- We use qpick[11] for our keyword matching requirements. It is built to be both memory and space efficient, while being able to scale to billions of queries.

Granne and qpick have been open-sourced under MIT and GPL-2.0 licenses respectively, do check them out!

It is hard to summarize the infrastructural challenges in running a search engine in a small section of this post. We will have dedicated blog posts detailing our Kubernetes infrastructure which enables all the above soon, so do check them out!

Final Thoughts

While some of our early decisions allowed us to drastically reduce the effort required in setting up a web scale search engine, it should be clear by now that there are still a lot of moving parts which must work in unison to enable the final experience. The devil is in the details for each of these steps – topics like **language detection**, **adult content filtering**, **handling redirects** or **multilingual word embeddings** are challenging at web scale. We will have posts on more of the internals soon, but we would love to hear your thoughts on our approach. Feel free to share and discuss this post, we will try our best to answer your questions!

References

1. Alphabet Q3 2019 earnings release - report ↩
2. Statista search engine market share 2019 - link ↩
3. Web Crawler - Wikipedia - link ↩
4. Spamdexing - Wikipedia - link ↩
5. Deep Web - Wikipedia - link ↩
6. Web Search Query Log Downloads - link ↩
7. SentencePiece - GitHub - link ↩
8. Granne - GitHub - link ↩
9. Gradient Boosting - Wikipedia - link ↩
10. Keyvi - GitHub - link ↩

11. qpick - GitHub - link ↩

Share this article

RSS Onion DAT