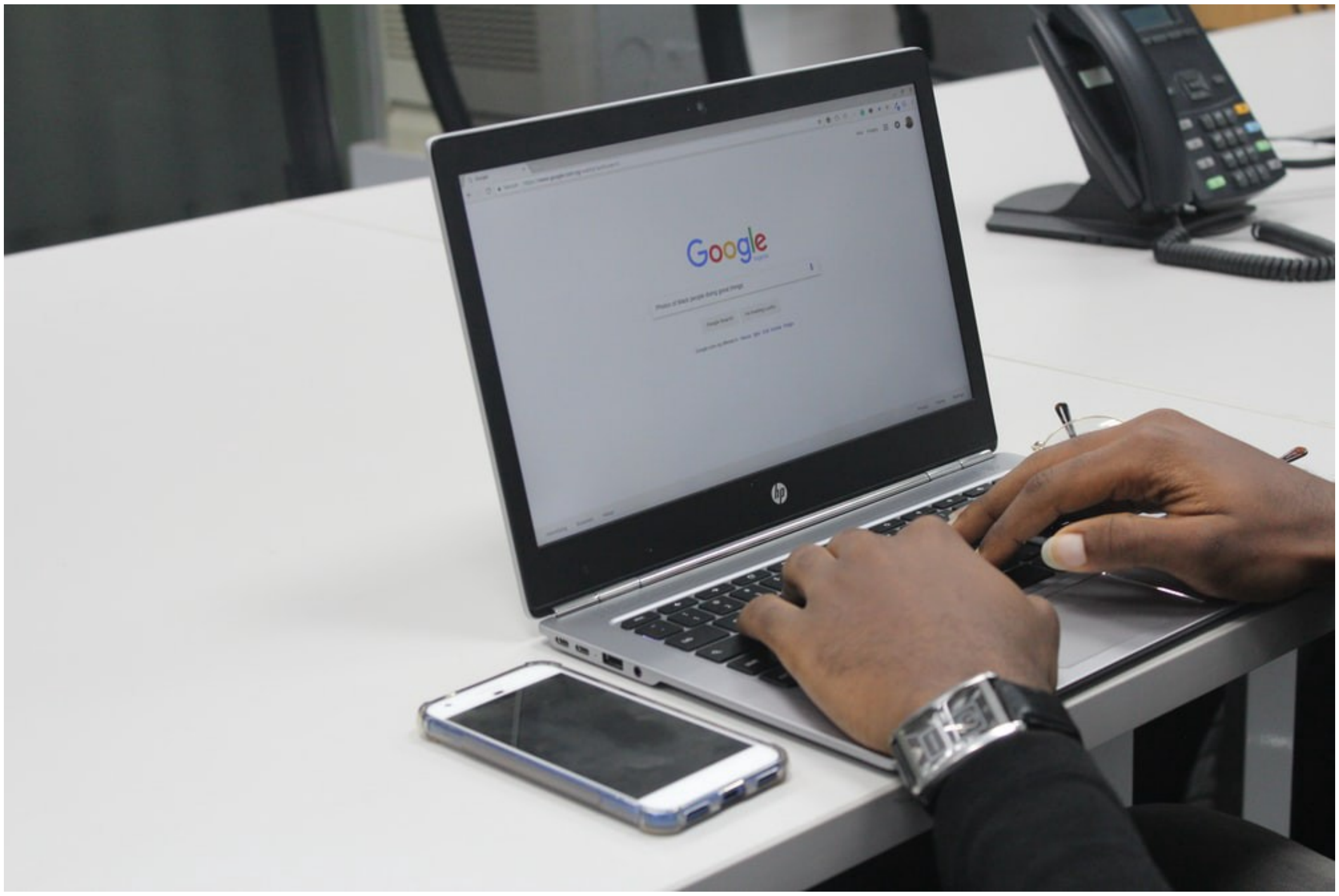
How I Would Design… A Search Engine!

A System Design Demonstration


Finding what you need, when you need it

Audience

This article is the next in my series of how I would design popular applications. It is recommended (although not entirely necessary) to read the previous posts I've helpfully compiled in a list here. We will expect a basic familiarity with architecture principles and AWS, but hopefully this post is approachable for most engineers.

Argument

Initially, let's look at our problem statement.

The System to Design

We would like to create a system similar to Google. Users should be able to enter a term and retrieve a list of web pages relevant to that term.

We also want to be able to highlight the term in the text, showing where it is located in the document.

The system should be low latency, highly available, resilient and scalable.

The Approach

We have a standard approach to system design which is explained more thoroughly in the article here. However the steps are summarised below:

1. **Requirements clarification:** Making sure we have all the information before starting. This may include how many requests or users we are expecting.
2. **Back of the envelope estimation:** Doing some quick calculations to gauge the necessary system performance. For example, how much storage or bandwidth do we need?
3. **System interface design:** What will our system look like from the outside, how will people interact with it? Generally this is the API contract.
4. **Data model design:** What our data will look like when we store it. At this point we could be thinking about relational vs non-relational models.
5. **Logical design:** Fitting it together in a rough system! At this point I'm thinking at a level of 'how would I explain my idea to someone who knows nothing about tech?'
6. **Physical design:** Now we start worrying about servers, programming languages and the implementation details. We can superimpose these on top of the logical design.
7. **Identify and resolve bottlenecks:** At this stage we will have a working system! We now refine the design.

With that said, let's get stuck in!

Requirements Clarification

The questions I would be asking would be: how many searches we are expecting a day, how big our documents are, how many documents we're expected to search, the average number of documents in a result, and traffic patterns (do we have lots of common searches, or a peak time for searching).

Let’s say we expect a billion searches a day, and have a billion documents of size `100KB`. Our traffic patterns are fairly consistent and we return 10 documents a search.

Back of the Envelope Estimation

If we have a billion documents of size `100KB` this is the same as `100,000,000,000 KB` of data to store, or `100TB`, which is a lot. Equally, a billion searches a day is the same as around 11,600 searches a second. If a search returns 10 documents, this is the same as `1MB` of data, so we’re sending around a gigabyte of data a second.

We can see it’s a fairly high traffic system!

System Interface Design

Now we need to decide how we would like to interact with our service. As we will see in our later section, we can assume that documents to return are already fed to us, we only need to design the search endpoint.

For this we can have a single endpoint `/search?query=<query text>` which returns a `200` response and a body similar to the below.

Error responses will show the regular `4xx` and `5xx` codes.

Data Model Design

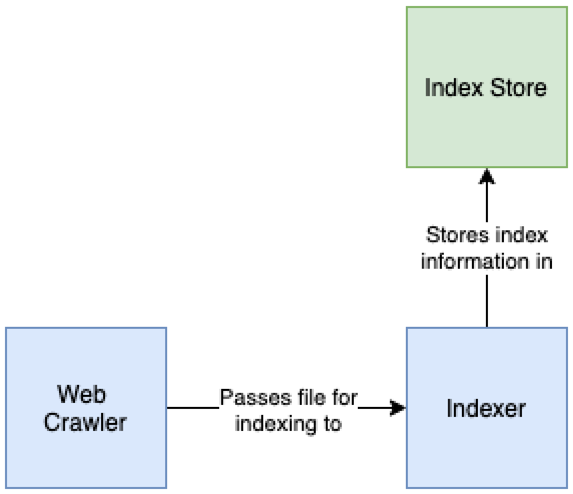
The meat of the problem lies in how we model our data. A large component of this section is taken from the video here.

A naive approach would see us iterate through our collection of documents looking for a term each time we search. When we find the term we would add our document to our result, then return the list. However, each search requires us to look at all of our documents, which is very slow.

A quicker solution is to do some pre-processing and use an index. Our service will trawl the documents, find the terms within the document, then create a map of terms to documents. This means when we search for a term we only need to find it in a single list!

So how do we create this map?

We can assume our system is fed into by a web crawler. If you aren't familiar with web crawlers you can read up using my article here. This means our system will receive a steady flow of documents that require indexing.

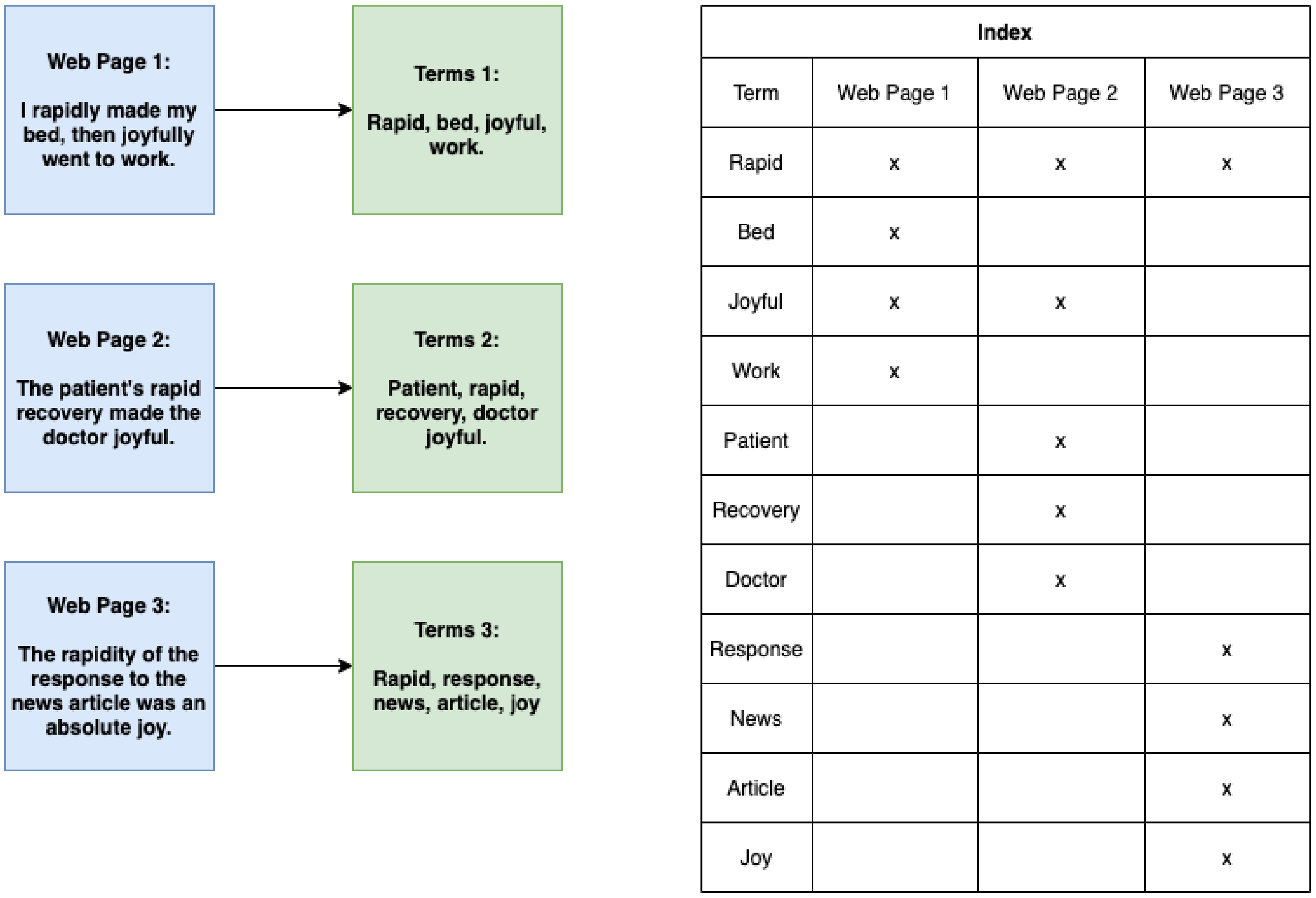


Simplified example system

Initially we need to do some processing of our documents. We will strip out all of the HTML tags, image links and technical components of the page, as we won't need these for searching.

We then need to do some additional work. We convert all words to lower case, carry out stemming and lemmatisation (converting words like rapid, rapidly, rapidity to just rapid for simplicity) and remove stop words (the, and, I etc.). We then have a corpus of documents with meaningful terms.

From there we list all of the terms and mark off which web pages contain them.



Web Page 1: I rapidly made my bed, then joyfully went to work. → **Terms 1:** Rapid, bed, joyful, work.

Web Page 2: The patient's rapid recovery made the doctor joyful. → **Terms 2:** Patient, rapid, recovery, doctor joyful.

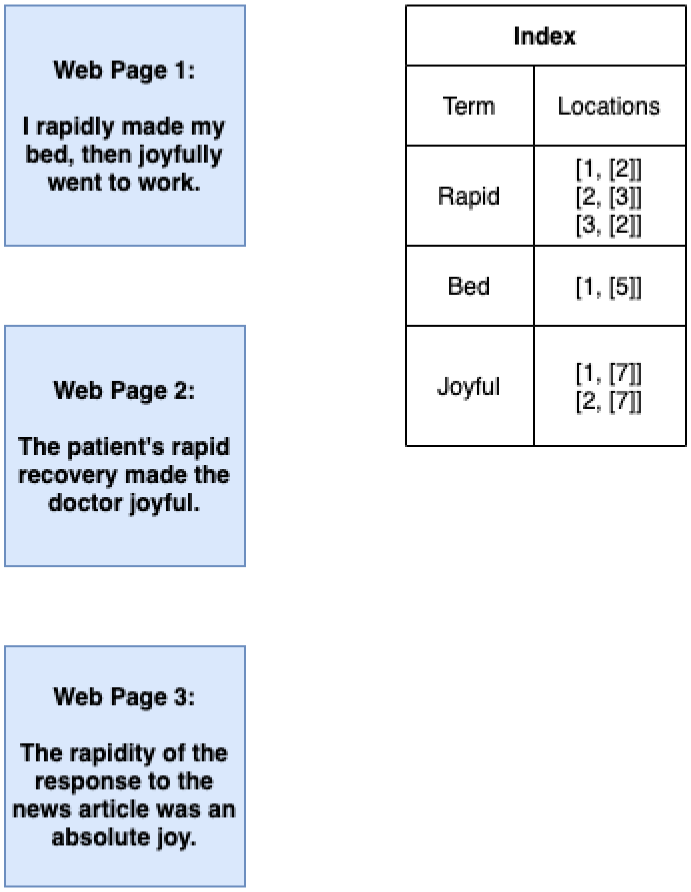
Web Page 3: The rapidity of the response to the news article was an absolute joy. → **Terms 3:** Rapid, response, news, article, joy

Index			
Term	Web Page 1	Web Page 2	Web Page 3
Rapid	x	x	x
Bed	x		
Joyful	x	x	
Work	x		
Patient		x	
Recovery		x	
Doctor		x	
Response			x
News			x
Article			x
Joy			x

Converting our web pages to indexes

There are a couple of ways we could store this information. The first is using a binary representation. For example rapid maps to `111` , bed maps to `100` , joyful maps to `110` . This is concise, but doesn't give us enough detail to do our required highlighting.

To highlight words we need to develop the idea a little further. We maintain a list of which documents our terms occur in, and where in the document they happen.



Web Page 1:

I rapidly made my bed, then joyfully went to work.

Web Page 2:

The patient's rapid recovery made the doctor joyful.

Web Page 3:

The rapidity of the response to the news article was an absolute joy.

Index	
Term	Locations
Rapid	[1, [2]] [2, [3]] [3, [2]]
Bed	[1, [5]]
Joyful	[1, [7]] [2, [7]]

Mapping our pages to terms and locations

The next thing we need to consider is how we combine terms in a search. For example, if someone searches ‘rapid joyful’ we might want to return the locations for pages 1 and 2, but not page 3 (conjunction, we need both words). Alternatively, for ‘joyful news’ we may want to return all pages (disjunction, we need either word). These can be implemented using an exclusion or union of results.

We can also do some calculations to see how relevant our result are. For example, the maximum distance between rapid and joyful in document one is 5, whereas in document 2 it’s 4, which suggests a higher relevancy.

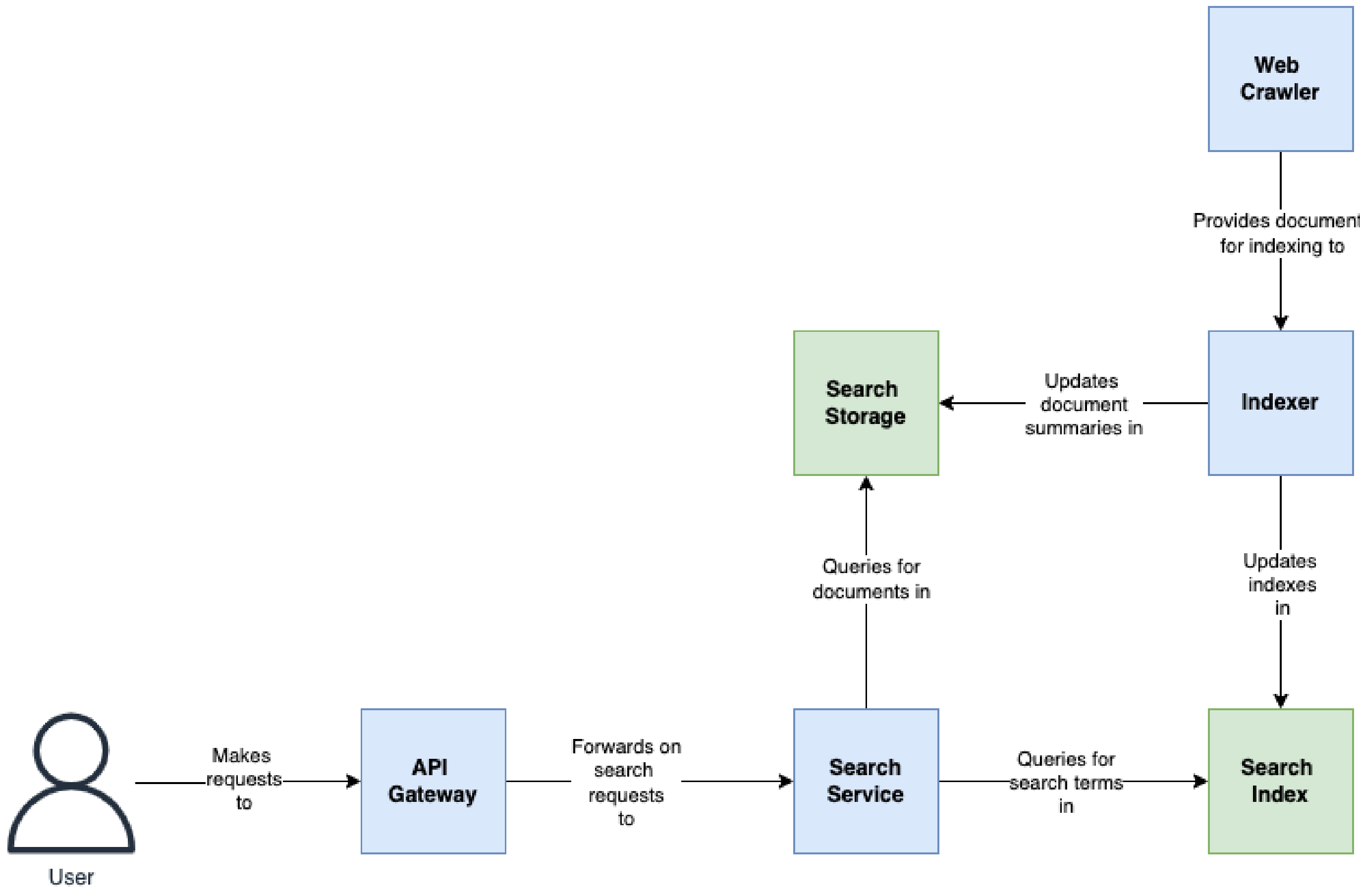
The other thing we could do is search for terms using the prefix. This is best implemented using a trie, which is fully covered in my articles here and here.

From all of the above it appears we are using a NoSQL based approach as we don’t need ACID properties, our traffic requires high availability, and it appears as if we can use a key (term) to value (documents and placements) store.

When we retrieve the documents themselves we can do something similar using a key (document Id) to value (document information as specified in the system interface section).

Logical Design

A basic logical design is contained below.



Basic search service logical design

We will assume we don't need to worry about the client application. Our user makes a request to our API Gateway, which we will use for throttling and other functional tasks.

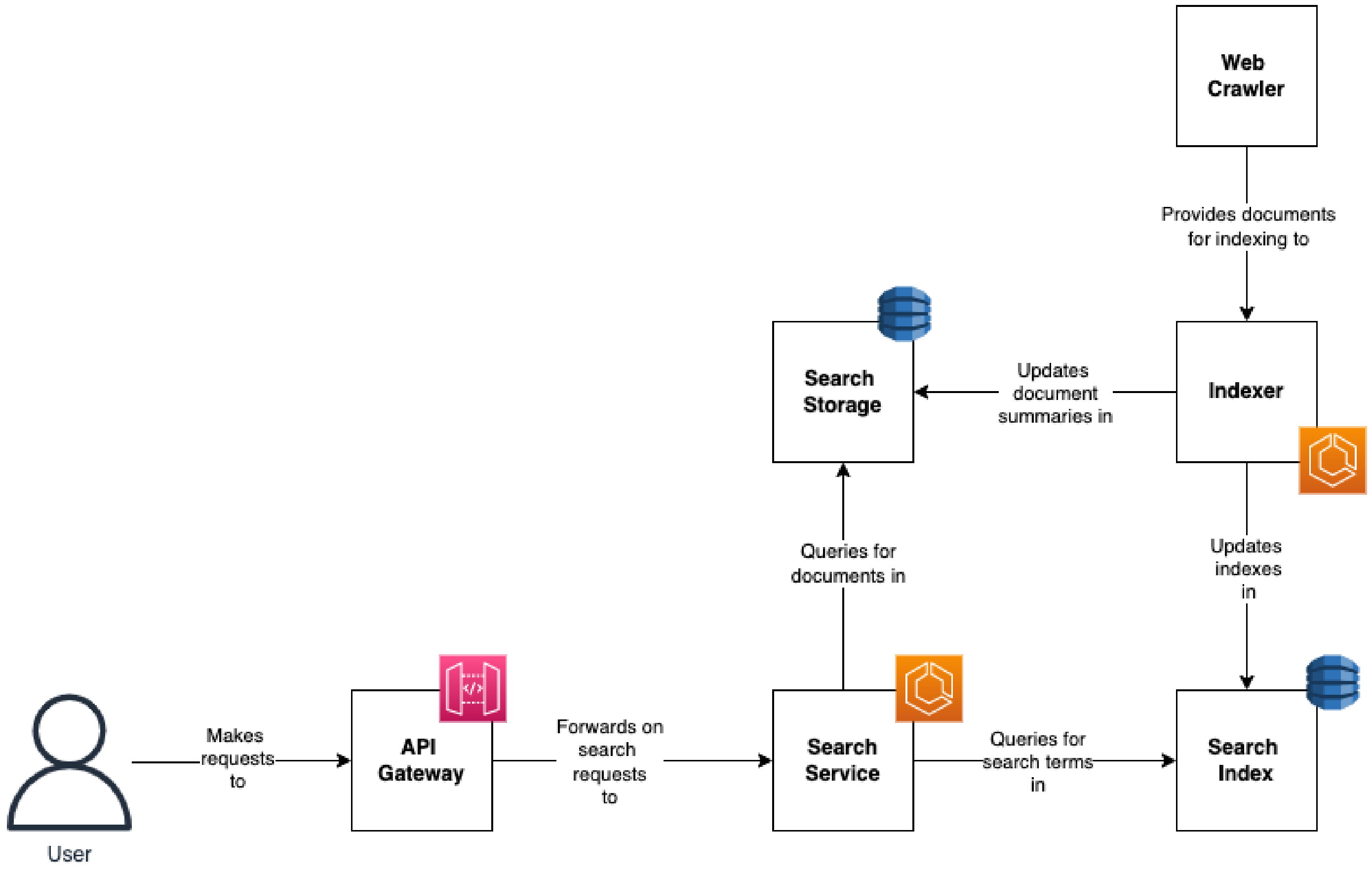
This forwards search requests to our search service. The search service has two responsibilities. The first is querying the search index to retrieve the list of documents and placements. The second is querying the search storage to retrieve the document summaries and aggregating them with the index results.

Going on behind the scenes is the indexer. This is fed into by a web crawler which is responsible for trawling the internet for documents. The indexer extracts the relevant information and stores them in the search index and search storage to be queried later.

Note, we could have used a single table design and stored our indexes and search information in the same table, however it is conceptually easier to understand them as two separate entities.

Physical Design

We can then transpose that into a physical design.



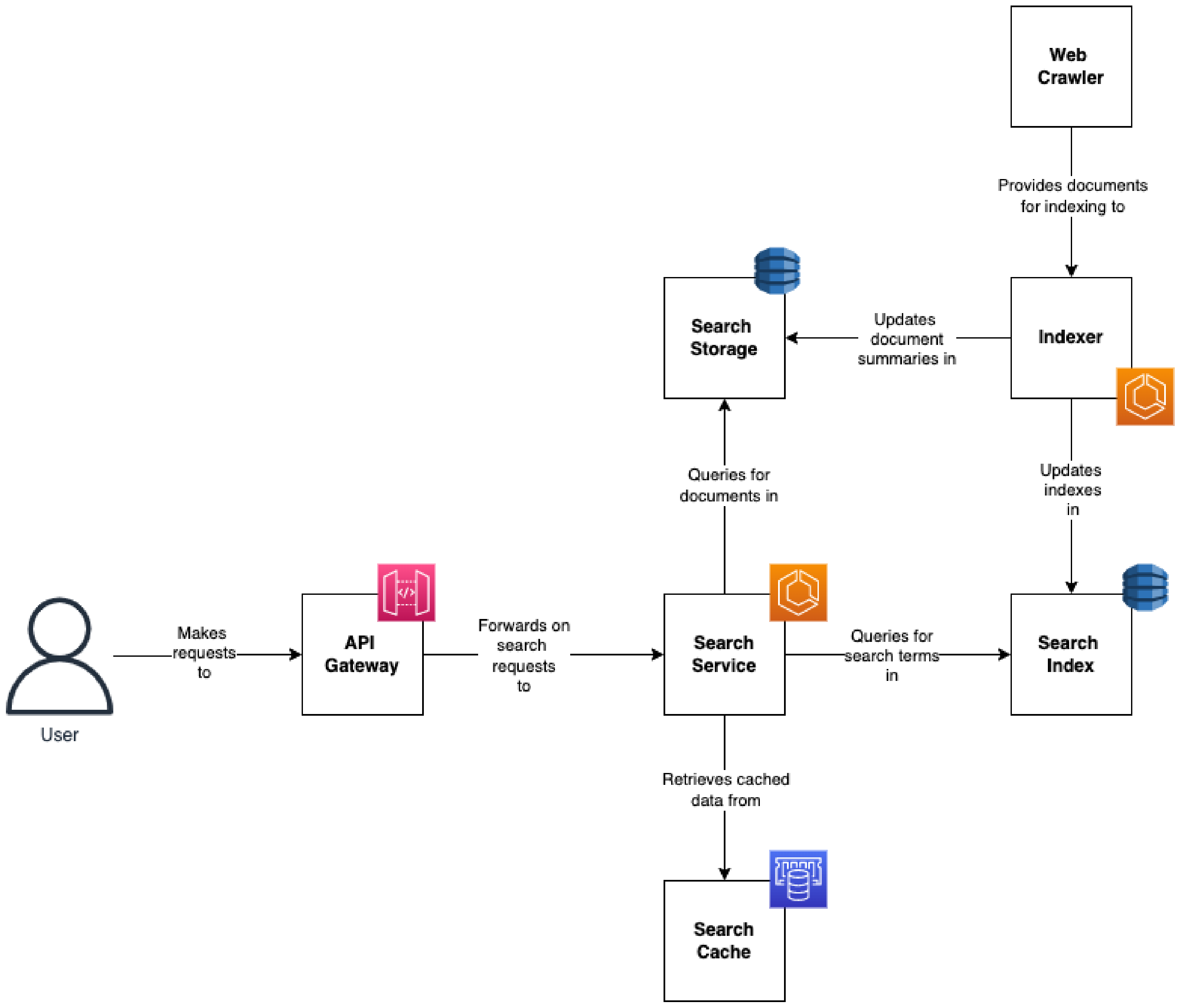
A basic physical design

In this physical design our API Gateway uses AWS API Gateway. It then forwards requests on to an ECS-based service to do the searching of indexes and information. We could have used a Lambda, however as there are so many requests it doesn't make sense to have to start one up per request, we may as well have a scalable solution already in place.

The storage solutions both use DynamoDb, which provide scalable, sharded data storage with constant return times. The indexer itself also sits on ECS.

Identify and Resolve Bottlenecks

The main bottleneck we may have is hot data, or terms that are queried often. To resolve this we can use a caching system. For this we could cache at the API Gateway or Search Service level. We will pick the Search Service. We can use a Least Recently Used or Least Frequently Used eviction policy.


Web Crawler
Provides documents for indexing to
Search Storage
Updates document summaries in
Indexer
Queries for documents in
Updates indexes in
User
Makes requests to
API Gateway
Forwards on search requests to
Search Service
Queries for search terms in
Search Index
Retrieves cached data from
Search Cache

Search service with cache

Conclusion

In conclusion we have summarised how one might go about designing a search service. There are already plenty of implementations (OpenSearch and ElasticSearch), but understanding it from the ground up will be helpful when you go and put them to use!