# Lucene Script

## Introduce yourself

Hey I’m Chris, A software engineer in the Routemaster team. Thank you for coming to this talk on Searching with Lucene. You may be wondering what Lucene is and how it applies to Trainline. Well, before we get into that let me tell you a bit about how we came to use it. There will be a Q&A at the end and please feel free to leave questions in the chat.

## The Frame

Last year there was a drive to bring the service that powers the station picker, which you can see on the right and I’m sure you’re all familiar with, into the backend. This was previously a service called Locations-Pot that was written in typescript. Locations-POT used AWS CloudSearch to search through the locations and then ordered them due set of priorities, for instance, an exact match or if it was a City.  
  
While this was a solution that worked for the current purposes, it lacked some flexibility, incurred a cost per search and many of the original developers that worked on the project have now left Trainline.

We wanted to bring this service in line with others by hosting it on the backend. Enter ReferenceDataSearch, a new service written in .NET to replace Locations-POT.

In the process of this we wanted to see what other options were out there to remove the dependency on AWS CloudSearch and therefore minimise costs, decrease latency as well as becoming more platform agnostic. We considered a few options away from CloudSearch, one of which was to write a lookup in house before eventually discovering Lucene.

## What is Lucene?

Apache Lucene is a free and open-source search engine software library; it was originally written in Java but has been ported to many other languages including C#, C++ and Ruby.

Lucene can be used for many kinds of queries which include full ranked text search, fuzzy matching for when a user makes a typo and spatial search for co-ordinates. A huge benefit of Lucene is its small memory footprint, with some indices being 20-30% of the size of the original text documents.

Lucene also supports incremental indexing meaning you can add to the index as you search as well as the ability to search on multiple indices at once.

This in combination with Lucene’s open source and extensible nature has enabled many community developed plugins and adoption all over the web, including at Twitter, Elastic and of course Trainline.

## How does Lucene work?

So, now you know a little bit about what Lucene is, let’s dive into some more detail about how it performs its queries.

*Read fundamental concepts*

On the right you can see we have an index that contains 2 documents, each document contains two fields, fact and submitter. Each of these fields in the document contains a set of terms. Reading from the first document you can see that there is a field named Fact, with the sequence of terms “Potato contains vitamin B6”

**Inverted index**

Next, we must understand how Lucene stores the information that it needs to lookup. To do this Lucene uses the concept of inverted indices where a token is associated with several document ids.

In the green box we see a list of document ids and the sequence of terms associated with the fact field from the previous slide. When this field is indexed, Lucene uses an analyser in order to generate tokens from the sequence of terms in the field.   
  
Analysers can have many different implementations and the one shown here is not applicable to all cases. Passing the sequence of terms through the analyser here results in the set of tokens in the blue box. You can see that the analyser has made all terms completely lower case and removed some terms be and with.   
  
These are known as stop words in Lucene, they are the most common words in any language (like articles, prepositions, pronouns, conjunctions). Removing them means we get more meaningful and speedy results as well as saving space. Lucene has lists of stop words in several languages and also allows you to add any words that might be required for your data set.  
  
Another thing that you can see that the Analyser has done is remove the pluralisation of the words “contains” and “potatoes”. This process is called stemming and allows a word to be searched by either its singular or plural form while retrieving the same result.

Once the analyser has generated the list of tokens, they are added to the inverted index which you can see in the red box. Each entry in the inverted index contains an index id, the token the entry refers too and a list of all document ids which contain the token.

For instance, here we can see that the token “potato” is contained in both documents 0 and 1 where as vodka is only mentioned in the document with id 1.

So now let’s move on to how Lucene performs a query against this index.

**Search**

A user or application will give us some query string, in this case that string is potato vodka. This string is then passed through the analyser in order to generate a set of tokens that can be used for querying.

The analyser is usually the same as the one used for indexing but not in all cases (for instance for when we use synonyms, we only want to do it on the index side. An example is aubergine and eggplant).

So now that we have our two tokens, “potato” and “vodka”, lets form some queries with them. You can see in the blue box that we have a query potato and vodka and in the red we have potato or vodka. These are two queries using the same tokens that produce different results.

We can see there are two tokens in the inverted index that match here.

**PRESS FOR ARROWS**

For the blue AND query we would only return document 1 as it contains both terms. However the red OR query will return both documents 0 and 1 as they both fit the criteria.

## Live coding demo

**INDEX**

Now that we understand the basic concepts of the Lucene engine, I think it’s right that we see it in action.

**Ask if people can see the code.**

Here I have a simple console application that uses a user input to search a lucene index and present the results back to the user. I can share this repo along with the slides later.

Firstly lets look at how we can create an index directory then and add documents to it. The directory here is a RAM directory so that it is stored in memory but you can also store on disk or database. We then define a writer that can be used to write files to this index.

This is also where we can specify an analyser for our fields, but don’t worry about that yet as we will cover that soon.

Once we have these two things, it is easy to create documents and add a set of fields to it. You can see here that have a document where we have added both a fact and submitter field, these are the same as on the slide deck.

We can also choose to store a field when we create an document, this allows us to retrieve that value later on. This has both memory and performance implications so should be used wisely, normally for some essential information like database key.

We can then write the document to the index and keep a reference to the document itself.

**Search**

From the search side we can see that we open up the same index directory as we created in the previous class and use it to create an index searcher that we can use to query.

The next step is to get the terms from the query string so that we can perform queries. I have extracted this into a separate class here for ease.

**PatternTokenizer**

You can see that the same analyser is used on the search side as the indexing side in this case, however that does not run true for all applications.

We then pass the query string into the analyser in order to retrieve a token stream. We also must pass a field name as some analysers use a “by field” representation, meaning the same string will be tokenised in different ways depending on the field they are destined for.

The token stream here works like any other and we are able to get the tokens for the search from it by yielding as normal.

**Search**

Once you have the list of terms you write queries with them. In this initial case we are just taking the first token and forming a term query for the “Fact” field. This query matches exactly the given term in the index. The query is then passed into the index searcher along with a max number of documents to retrieve.   
  
After execution, the documents are returned in an array that we can iterate over. You can see here that because we stored the “Fact” field when we were building our directory, we are able to retrieve it here.

**Analyser**

Now before we look at what our code does upon execution, let’s look at the analyser.

To start with, we are using a simple Analyser base class.

We must override the CreateComponents method which creates the TokenStream. Here we are using a standard one that does nothing but create tokens by delimiting on white space.

**First search**

For this initial search we only expect to get exact term matches, this includes capitalisation and plurals.

We can see that typing “potato” gives us back no results. This is because at the moment we are not normalising the text from the input of the document to lower case or stemming the plural in “potatoes”. If we type “Potato” it gives us back the first fact, “potatoes” gives us back the second fact.

Now lets see how we can make this a better searching experience.

**Add lower case**

To do this we can add these token filters one at a time to get better results from our queries. Let’s start by adding a lower case token filter so that all tokens are converted to lower case before they are added to the index and before they are used in search queries.

Now when we run the program and search “potato”, we get the first fact but still not the second as it is pluralised. Also, if we were to query “vodka” we would get the second fact, proving the normalisation works.

So let’s look at how we can keep improving and remove those common stop words. You see if I search here with “with” we would still get the result back, now as that is a very common word it can mess with search results and dimmish user experience.

**Add stopwords**  
  
Here you can see that we pass the STOP\_WORDS\_SET into the analyser. This is the set of standard English stop words provided by Lucene. There are implementations in several different languages that can be used. It contains words like “be” and “with”.

We must apply the lower case filter before using this and pass that token stream into the filter along with the m\_stopwords which is inherited from the base class and populated by the constructor we used earlier.

USE WITH and BE

Now let’s look at removing the pluralisation of words so that we can get results back for both facts when we search potato.

**Add stemming**

Here we pass the token stream resulting from the last stage into the PorterStemFilter. PorterStemming is an algorithm that removes pluralisation as well as many other things in order to improve searches. In many cases it might be too powerful, however there are many more stemming filters available in Lucene that support multiple languages.

Now we can see that searching “potato” gives us back both facts and “contain” gives us back the first. We can also see that “contained” and “container” provide results, this is because the powerful porter stemming algorithm enables us to search not only on pluralisation’s but on tenses too.

**Now do multi term OR**

So now let’s use this analyser and switch up our queries. Firstly let’s start by looking at Boolean queries   
  
Here we have a list of the term queries generated by the tokens we received from the analyser, this is similar to how we were originally generating a single query but for multiple terms. We can chain these queries together using a BooleanQuery.

Here you can add the queries to a Boolean query with either a Must, Should or MustNot. Referring back to the slide show we can see that we wanted for the blue box, potato and vodka, for the red box we wanted potato or vodka.  
  
Using the MUST keyword we can say both terms must occur, therefore when we search potato vodka, only the second fact comes back.  
  
However using the SHOULD keyword means either term should occur but at least one of them must in order to match a document. Searching again we see that both facts come back.

**Prefix search**

Another cool type of query that really enables a better user experience is the prefix query. This allows a user to only type part of a word in before they get a result.

This is again created in a similar fashion to the term query except using PrefixQuery in it’s place.

Using pot for potato we can see that we get both results back even though the term only matches the prefix.

**Fuzzy Search**

Lastly, we know that some users may have difficulty spelling, especially in our application where some place names are extremely hard to spell. If you need an example, then you need not look further than pretty much anywhere in Wales.

Fuzzy search works by finding the number of edits taken to transform one word into another, this is a called the levenshtein distance. In our example you can see that we allow for up to 2 edits.

Using this fuzzy query we can see that misspelling potato now gives us results for the two facts.

Potata – 1 edit

Patata – 2 edits

Patota – 3 edits

**Summary**

So in summary, Apache Lucene is a super-fast and customizable tool that can be used for full text searching. It is maintained by the Apache Software Foundation and has a large developer community, so you know there is plenty of support when needed.

It has allowed us to improve the search experience by utilising it in the Reference Data Search. Recent improvements include Fuzzy searching, Searching by translations and searching by synonyms or colloquial names. For example for searches to stations like Gare Du Nord where it can be known by both Nord and Gare Du Nord.

Thanks for listening. I guess we can move on to questions if there are any