Sub-second search on CanLII

Marc-André Morissette*, Daniel Shane**
**Director of Technology, Lexum, morissette@lexum.com*
***Senior developer, Lexum, shaned@lexum.com*

Abstract. Search engines are at the heart of a Legal Information Institute. They are the principal mean by which its users access the information it offers. It is therefore of critical importance that this tool be as efficient as possible. A key component of this "efficiency" metric comes from the time it takes for the search engine to compute a query and return results to the user. This article looks at several ways in which search engine performance can be improved. It investigates the components of server performance that drive search engine performance as well as some information that can be pre-computed by modern search engine to minimize the cost of complexity associated with large collections of data.

Keywords: Information Research, Performance, Bigrams, Shingles, Lucene, CanLII, Disk Performance

1. Introduction

In 2008, several hundred members of various law societies in Canada were surveyed about CanLII. In addition to questions about how they perceived the free service, a number of questions about what they felt could be improved were asked. The answers were not surprising. One, CanLII needs even more content: more courts, more tribunals and older material. Often, respondents were kind enough to be specific about which courts or tribunals to target. Two, it would be nice if it were easier to search. How to get there however, few respondents could say.

Search is hard. For all the talk about synonyms and concept based searches in the scientific field of information research, very little progress has been made. In essence, what worked forty years ago: matching query terms to identical terms in documents is still the basis of modern search algorithms. Even Google, at its core, still seems to mostly rely on this. Except for two advances, in our opinion, improvements in search relevance since then were not revolutionary, merely evolutionary. Those two exceptions are: the use of hyperlink ranking for the Web (Page 1998, Qin, 2005) and the improvements in search speed. This article is about the latter.

Thirty-five years ago, a search in a legal database of a few megabytes could take minutes to complete. To deal with this, users were given complex and advanced query syntaxes, told to carefully craft their queries and made to pay a hefty price if they had to search again.

Nowadays, we expect a Google search into billions of documents to complete under a second and for fractions of a penny. This has completely changed people's relationship with search engines. Instead of performing one search, people jot a few words, assess what's missing from the first few results, tweak their query and repeat the process. We have witnessed it in our logs: keyword searches are refined over and over until users finally find what they are looking for. For keyword searches, the median number of iterations is 4.

For years, CanLII has always been proud to say that its search engine is fast and will return results in less than a second. However, as the number of documents in CanLII's databases grew, this became less and less true. The issue was the growing size of our databases. By the time this article is made public, CanLII will publish more than a million court decisions and over 100,000 legislative texts. This represents more that 3.5 billion words or 13 million pages of unique legislative texts. And of course, these totals keep growing. Given that the performance of a search engine generally degrades linearly as the amount of indexed information grows, the slow degradation of our search engine's performance was not surprising.

In addition to this, CanLII recently added a RSS alert function to its search engine, enabling users to receive alerts whenever a new decision matches a specific search query. RSS news readers are implemented in such a way that every once in a while (generally between every few minutes to a few times a day), they will ask the Web Server responsible for the feed for an updated list of recent items. Knowing this, we were worried that if the functionality became popular, the amount of queries served would increase dramatically and degrade search performance for all users.

Late last year, we thus endeavoured to improve the performance of our search engine. We gave ourselves the following goals:

1. a search must be answered within one second, even in complex cases; a simple (two or three words) query be answered even faster; and
2. a significant increase in search traffic should not become an issue for our servers.

2. How a Search Engine Works

In order to discuss some of the performance issues behind our search engine, one must first understand how it works. In this section, we present the design behind most modern search engines and how Lexum's search engine differs from the norm.

2.1. GENERAL SEARCH ENGINE THEORY

Most if not all search engines work in the same way (Baeza-Yates, 1999). First, an *indexer* processes the document to be searched in order to create a *search index*. You may think of this *index* as a database of information that will allow the search engine to quickly build a result set for a given query. The

indexer is simply the software that will create and maintain this database given a collection of documents.

Second, given a certain *search query*, a *searcher* will read this *index* and provide results based on the content of the index. The concept is that using the index to answer the query will be faster than reading all of the indexed documents for every query.

Although the indexes used by modern search engines vary greatly, they generally are based on the concept of an *inverted index*. Inverted indexes are analogous to dictionaries in that they contain a long list of words sorted alphabetically. This list of words comes from the indexed documents. However, in lieu of a definition, an inverted index contains a list of documents containing the word. In addition, for every document containing the term, a list of all positions occupied by the term in every document is written. This design makes it easy to locate one word and then locate every occurrence of that word in every document.

As an example, consider two simple documents:

- Document 1: The quick brown fox jumped
- Document 2: The fox and the dog sleep

An *inverted index* for these two documents would look like the following:

Word	Doc. Freq.	Occurrences
And	1	(Doc=2;Freq=1;Pos=3)
Brown	1	(Doc=1;Freq=1;Pos=3)
Dog	1	(Doc=2;Freq=1;Pos=5)
Fox	2	(Doc=1;Freq=1;Pos=4) (Doc=2;Freq=1;Pos=2)
Jumps	1	(Doc=1;Freq=1;Pos=5)
Quick	1	(Doc=1;Freq=1;Pos=2)
Sleep	1	(Doc=2;Freq=1;Pos=6)
The	2	(Doc=1;Freq=1;Pos=1) (Doc=2;Freq=2;Pos=1,4)

Table 1: An example of an inverted index

In this table, *Doc. Freq.* is the number of documents that contain the specified word and *Occurrences* describes for each document the number of times the word is present as well as in which positions it occurs.

The *searcher* uses the information found in the inverted index to compute a score *S* for each document *d* in the index given a query *Q*. This score is a measure to compute how relevant a document is to the query. The higher the score, the higher the relevance. Although the method by which this score is computed is a very active area of research where all sorts of interesting ideas have been tried, the sad fact of the matter is that most search engine still rely on the Vector Space Model, a model that was developed in 1975 by G. Salton,

(Salton, 1975) the newer models only yielding modest improvements[1] despite often being much more complex.

The model essentially relies on two measures:

1. Term frequency: the number of times a search term is found in the document; the underlying assumption being that if a term occurs more often, the document is more relevant to that concept

2. Inverted Document Frequency: a measure of the "uniqueness" of the term by computing in how few documents the term occurs; the underlying assumption being that if a term is rare, each of its occurrence should contribute more to the score. Think of the query *syncopation sound*: the salient term of the query is *syncopation* so it should primarily determine the relevance of a document. However, because *sound* is much more likely to be found in the collection of documents, without the Inverted Document Frequency component, the score of most documents would primarily be determined by *sound*.

Although the formula typically used is much more complex, the basic formula used by most Vector Space Model search engines resembles this one:

$$\text{Score}(d) = \sum_{t \in Q} \left[\frac{n_{t,d}}{\sum_{t' \in d} n_{t',d}} * \log\left(\frac{|D|}{1 + |\{d' \in D : t \in d'\}|} \right) \right]^2$$

where

- *D* is the collection of all indexed documents;
- *d* is specific document in *D*;
- *Q* is the query;
- *t* is a term of the query *Q*; and
- *t'* is a word in document *d.*

The part within the bracket that is left of the log function is the *term frequency* component. The part to the right of the log function is the *inverted document frequency* component.

To understand the performance implication of this model, let's look at a simple query over our two example documents: *fox sleep*. The part of the inverted table used to compute the model follows:

[1] As we have said, an exception to this have been Web search engines which have benefitted from newer models that exploit the hyperlinked nature of the Web. Unfortunately, these improvements do not translate well to regular document collection. Attempts to apply the same models to legislative citations have so far met with limited success.

Word	Doc. Freq.	Occurrences
Fox	2	(Doc=1;Freq=1;Pos=4) (Doc=2;Freq=1;Pos=2)
Sleep	1	(Doc=2;Freq=1;Pos=6)

Table 2: the inverted index portion necessary for the query

The inverted document frequency part of the formula can be calculated once for each term of the query. It is calculated using the Doc. Freq. column of the inverted index.

Then, for each document that contains one of the searched terms (an occurrence tuple), the Term Frequency part of the formula can be computed from the Freq component of the occurrence. The score of the document can be calculated from the Term Frequencies and the Inverted Document Frequencies.

Therefore, from a complexity standpoint, the time required to return results to the user is a function of:

1. the number of query terms; and
2. the number of documents containing these query terms, which on average means the number of documents in the collection.

2.2. PHRASES

A factor that may increase the complexity of a query is the use of phrase operators or proximity operators. In most search engines based on the Vector Space Model, phrases are treated as if they were a single term. Their frequency and inverted document frequency can be calculated by how many times the phrase appears in the document and how many documents contain the phrase.

However, in order to determine whether a sequence of terms (a phrase) occurs in a document, their position must be fetched from the inverted index and compared with one another.

As an example, consider the query *"the fox"* over our two example documents. The relevant part of the inverted index follows:

Word	Doc. Freq.	Occurrences
Fox	2	(Doc=1;Freq=1;Pos=4) (Doc=2;Freq=1;Pos=2)
The	2	(Doc=1;Freq=1;Pos=1) (Doc=2;Freq=2;Pos=1,4)

Table 3: the inverted index portion for the words *the* and *fox*

The criteria for a phrase occurrence (each occurrence will contribute to the score of the document) for the query is that

1. the word *the* and *fox* both be found in the same document;
2. that there be at least one occurrence of the word *the* occurring in position p and an occurrence of *fox* occurring in position $p+1$.

In the preceding case, consulting the inverted index, it becomes clear that document 2 satisfies the condition while document 1 does not.

From a complexity standpoint, phrase operators are much more demanding than simple AND/OR operators as they require a position comparison. In the case of phrases, the time required to return results to the user is a function of:

1. the number of phrase terms;
2. the number of documents containing all phrase terms, which, in the average case is congruent to the number of documents in the collection; and
3. the number of times each term occurs in the documents where both terms are present.

2.3. LEXUM'S SEARCH ENGINE

The model used by Lexum is a variation of the Vector Space Model (Morissette, 2007). It was modified to better deal with document length bias, deal with some corner cases, add stemming, handle boolean queries and natural language queries. Although the details of how this is done is beyond the scope of this article, from a complexity standpoint, the only part that matters is how it deals with natural language queries (queries with no operator).

Several years ago, Lexum proceeded to analyze the set of queries made with Lexum's search engine with a view to improve its behavior. One of the findings of this analysis was that most users don't bother using boolean operators, even in cases where they would significantly improve their results. Consider, for example the query *Criminal Code Section 16*. This query returns 33,052 results on CanLII, most of them not relevant. However, by applying the proper operators *"Criminal Code" /s "section 16"*, 179 mostly relevant results are returned.

We therefore endeavored to better understand how to handle queries with no operators. Our analysis led to three conclusions.

First, we were made to understand that our users prefer to have a clear and understandable rule that governs whether or not a document is included in the results for a query. An example would be that the default operator is AND. However, the algorithm that governs the ranking within these results does not have to meet such a requirement i.e. it is not necessary to rank as if the AND operator was used.

Second, we discovered that in order to conceivably offer appropriate results in the vast majority of natural language queries, it is better to only consider documents that contain all of the search terms. This is because the vast majority of users input their queries in the form of a set of keywords and very few of these keywords are synonyms. In fact, when users bother to find a list

of synonyms for their query, they most often use the proper boolean OR operator. An representative example of a typical query without operator with would be *infection hospital negligence*. Although it is conceivable that a document missing one of the words could be relevant, it is much more probable that documents introduced by relaxing that constraint would be less relevant to the query.

Third, in more than half of the natural language queries inspected, we noticed that the introduction of a phrase operator for a subset of the keywords would have considerably improved the quality of the expected results. Queries like "*Marc Morissette"* or "*R. v. Oakes"* or "*criminal code" "section 16"* return better results when the phrase operator is applied.

These three conclusions led us to devise a new algorithm called *Subphrase Scorer* to score documents against multi-word queries that contain no operator. This Subphrase Scorer simply returns all the documents that contain every term in the query. However, the ranking is performed by considering every possible permutation of the phrase operator and assigning a probability to each permutation. The probability for every permutation comes from a machine learning algorithm that was trained on a set of properly quoted queries sampled from our query logs. For every permutation, a score consistent with the given phrase operator permutation and then weighted by the probability of the permutation.

As an example, consider the query *Criminal Code section 16*. The possible permutations with its associated probabilities are the following:

Permutation	**Probability**	**Score computed using simple Vector Space Model for an example document**	**Weighted score**
Criminal Code section 16	28%	0.45	0.126
"Criminal Code" section 16	20%	0.85	0.17
Criminal "Code section" 16	1%	0.9	0.009
Criminal Code "section 16"	17%	0.7	0.119
"Criminal Code section" 16	2%	0	0
Criminal "Code section 16"	1%	0	0
"Criminal Code" "section 16"	35%	0.9	0.315
"Criminal Code section 16"	1%	0	0
Total	**100%**	**Sum (Score assigned by Subphrase Query)**	**0.739**

Table 4: All possible permutations of *Criminal Code section 16* along with an example of how score would be weighted for a sample document

The third column and fourth columns provide an illustration of how the scores for a particular document are weighted.

The algorithm computes the scores for all variations in one pass, therefore the SubPhrase query is not as long to compute as the sum of all permutations.

However, from a standpoint of complexity and performance, the *Subphrase query* takes slightly more time to compute than a simple regular phrase query because some optimizations cannot be applied.

2.3. PERFORMANCE IN DETAIL

Now that the basics of the inner workings of Lexum's search engine are clear, it is possible to explain the factors that impact the performance of a search engine. To better illustrate the issues, let's return to our example query: *Criminal Code section 16*. In order to compute the score of all documents, one must look at all occurrences of every document where every term pair is present: *Criminal Code*, *Code Section*, and *Section 16*. In addition, frequency information for every single word will have to be taken into account. To compute that information, the appropriate occurences are read from disk into memory and the CPU performs the computations outlined before.

The time required to perform these operations depends on the size of the indexed collection. Let's look at those statistics for CanLII's current index.

Word	**Doc. Freq.**	**#Occurrences**	**Size of the occurences in inverted index**
Criminal	86,560	573,901	3 MB
Code	229,905	1,062,291	6 MB
Section	431,147	5,812,538	27 MB
16	597,170	1,745,383	12 MB

Table 5: Statistics from CanLII's indexes

What becomes immediately obvious is how much information needs to be read from disk to perform a single query: 48 MB for this particular example. Given that CanLII receives more than 2-3 queries per second in the afternoon during peaks, this puts a tremendous load on Lexum's disk arrays. We therefore had issues where performance during peaks would considerably decrease as disk arrays reached maximum throughput.

The other component of the performance equation is the amount of CPU cycles required to compare positions against each other, retrieve the frequencies and compute the aggregated scores of each document. For the example query, the number of algorithmic iterations required to compute all scores comes down to 7,157,000. That is the amount of comparisons of occurrence positions against each other.

Now this is all theory. In order to better understand how these statistics affected our search engine in the real world, we profiled the performance of the same query *Criminal Code Section 16* against an idle server.

Component	Time required	Notes
Basic overhead	120 ms	Amount of time required for all queries to perform basic search duties not related to the inverted index (constant)
Result display	350 ms	The amount of time required to fetch result information: titles, references, etc. (constant)
Fetch occurrences from disk	3000 ms	Variable. May reach 9 seconds for queries with many terms
Compute score	1830 ms	The amount of time required to match occurrences and compute the score. Variable. May reach 10 sec. for 10 term queries.
Total	**5.3 sec**	

Table 6: performance of our search engine over a simple 4 term query

Clearly disk access was a major problem. In addition, the complexity of the algorithm made it difficult to keep the system response time under one second for moderately complex queries.

3. Solutions and results

3.1. N-GRAM INDEXATION

As we have seen in the previous section, the biggest drain on performance is related to the fetching of the occurrence positions from disk and the comparison of the positions of sequential terms, especially when two frequently occurring terms occur next to each other.

In order to solve this issue, we elected to pre-compute some of that information by storing in the inverted index every word pair (also called a bigram) occurring in the collection. As an example, consider one document, with the following content: *The quick brown fox jumps*. The inverted index is now augmented to contain word pairs as well as individual words.

Word	Occurrences
Brown	(Doc=1;Freq=1;Pos=3)
Brown-Fox	(Doc=1;Freq=1;Pos=3)
Fox	(Doc=1;Freq=1;Pos=4)
Fox-Jumps	(Doc=1;Freq=1;Pos=4)
Jumps	(Doc=1;Freq=1;Pos=5)
Quick	(Doc=1;Freq=1;Pos=2)
Quick-Brown	(Doc=1;Freq=1;Pos=2)
The	(Doc=1;Freq=1;Pos=1)
The-Quick	(Doc=1;Freq=1;Pos=1)

Table 7: Inverted index containing bi-grams

The advantage of having this new information is that it is now possible to compute two-term phrase occurrences much faster. For example, to compute the query *"brown fox"*, only the frequencies present in the index for the term Brown-Fox are necessary. Positions are not necessary.

For phrases longer than two terms, one can adapt the algorithm previously used for the phrase operator to compare the positions of bigrams instead of comparing the positions of single terms. As an example, the query *"quick brown fox"* can be computed by retrieving every occurrence of Quick-Brown and Brown-Fox and apply the same criteria as for a phrase search.

For a query, such as "the quick brown fox", the algorithm can be further modified to take into account the 2 position jump between subsequent bigrams. The criteria for a phrase occurrence thus become

1. that the bigram *the-quick* and *brown-fox* both be found in the same document;
2. that there be at least one occurrence of the bigram *the-quick* occurring in position p and an occurrence of *brown-fox* occurring in position $p+2$.

Although the addition to the index of every bigram in the collection necessarily increases the size of the inverted index (more than doubling it) and the time required to create it, it dramatically reduces the time normally required to compute the results for a query. This improvement mostly comes from the fact that *the-quick* and *brown-fox* will generally occur much less often in a collection than *the*, *quick*, *brown* or *fox.*

As a real world example, let's consider once more the *Criminal Code Section 16* query on CanLII. The following table illustrates well how faster the new algorithm is with real world data.

Term	Without bigrams		With bigrams	
	Calculations	**Size on disk**	**Calculations**	**Size on disk**
Section	431,147	27 MB	431,147	1,7 MB
16	597,170	12 MB	597,170	2,4 MB
Criminal	85,560	3 MB	85,560	0,3 MB
Code	229,905	6 MB	229,905	0,9 MB
Section-16	4,446,400		16,683	0,3 MB
16-Criminal	644,938		375	0 MB
Criminal-Code	610,981		49,737	1,2 MB
Total	**7,151,011**	**48 MB**	**1,413,867**	**6,9 MB**

Table 8: Theoretical performance improvement due to bigrams

We see that on paper, the introduction of bigrams yields a 5-fold reduction in the number of calculations to perform as well as a 7-fold reduction in the amount of data to retrieve from disk.

In real world performance, here is how the bigram algorithm performs next to its predecessor. These tests were performed over several iterations.

Component	Without bigrams	With bigrams
Basic overhead	120 ms	140 ms
Result display	350 ms	350 ms
Fetch occurrences from disk	3000 ms	2000 ms
Compute score	1830 ms	400 ms
Total	**5.3 sec**	**2.9 sec**

Table 9: Real world performance improvement due to bigrams

We notice that although the performance improvement yielded by the new bigram algorithm for CPU computations is close to the theoretical value, the improvement in time required to fetch the data from disk falls short of the theory. We can only speculate that the amount of data to be fetched is not the only factor in disk performance. Indeed, mechanical disks are limited in other ways such as the number of non-contiguous pieces of information that can be retrieved per second.

3.1. BRUTE FORCE

Although the majority of queries processed by CanLII's search engine either contain no operator or contain simple phrase operators, a non-negligible portion of queries, usually performed by more advanced users, contain proximity operators. CanLII supports the /*n* (within n words, n being an integer), /p (within the same paragraph and /s (within the same sentence) operators.

These operators, like the phrase operator, also require the position of single words to be read and compared. Unfortunately, bigrams cannot improve the performance of such operators. In extreme cases, where many such proximity operators are used, the time required to compute a query may be prohibitive, generally on the same order than phrase operators without bigrams.

Take, for example, the following query: *equipment /p (ownership OR title) /p determination*. This query has the following performance characteristics:

Component	Performance
Basic overhead	140 ms
Result display	350 ms
Fetch occurrences from disk (including paragraph markers)	2 500 ms
Compute score	2 050 ms
Total	**5,0 sec**

Table 10: Real world performance of a proximity query

Unfortunately, as far as we know, very little can be done the performance of proximity operators. It was thus decided that brute force i.e. a server upgrade was necessary.

It was not clear how to proceed however. Upgrading the processors was easy: we selected a 12-core system with two X5680 Xeon processors; two of the most powerful processors available at the time.

The problem of disk performance, however, presented a dilemma. Our search server was connected to a 40 disk HP EVA 4000 Storage Area Network; basically a rather expensive bunch of networked disk that could handle several thousand disk operations per second. How could we improve on this without breaking the bank?

We considered adding more than one hundred gigabytes of memory to our search server in order to store the inverted index into memory. Unfortunately, the plan seemed to be very expensive and fraught with lengthy startup times between reboots and inverted index updates. Reading 100GB of data from disk to memory is a slow process.

We looked at solid state disks (disks that store information with flash memory) but could not find any that had the appropriate throughput until we learned of a specialized disk solution called the Fusion-IO IoDrive Duo SLC, basically a specialized flash drive capable of 260,000 disk operations per second (2600 times as much as a desktop drive and 50 times as much as our existing SAN).

Although the improvement yielded by the new processors was modest, the improvements afforded by the new disk system were nothing short of spectacular.

The following two tables, respectively detail the performance of two queries: *equipment /p (ownership OR title) /p determination* and *criminal code section 16*. They illustrate how important disk systems are to search engine performance:

Component	Old server	New server
Basic overhead	140 ms	30 ms
Result display	350 ms	270 ms
Fetch occurrences from disk	2 500 ms	negligible
Compute score	2 050 ms	1100 ms
Total	**2.8 sec**	**1,4 sec**

Table 11: Real world performance of *equipment /p (ownership OR title) /p determination*

Component	Old server (with bigrams)	New server (with bigrams)
Basic overhead	140 ms	30 ms
Result display	350 ms	270 ms
Fetch occurrences from disk	2000 ms	negligible
Compute score	400 ms	250 ms
Total	**2.8 sec**	**550 ms**

Table 12: Real world performance of *criminal code section 16*

4. Conclusion

In this article, we have looked at the factors that govern search engine performance. In particular, we have discovered that collection size and query complexity are the major factors that govern query performance.

We have introduced a technique that mitigates these factors for the vast majority of queries by pre-computing every bigram in the collection and inserting that bigram in the inverted index.

Furthermore, we have demonstrated that by focusing mostly on disk performance instead of processor performance, it is possible to drastically reduce the time required to compute the results of a query, whether bigrams are used or not.

As our collections grow and as we add complexity to our scoring formula in order to make it more efficient, there is not doubt that further improvements will be required to keep our search engine responsive.

One such improvement might be to break-up our inverted index into several smaller indexes and distribute searches across several different servers as others have done (Brin, 1998). A much less ambitious project would be to parallelize the computation of the scoring function such that all server cores can be used at once during the processing of a query.

References

Page, L., Brin, S., Motwani, R. and Winograd T. (1998) *The PageRank Citation Ranking: Bringing Order to the Web*. Technical report, Stanford Digital Library Technologies Project.

Qin, T., Liu, T. Y., Zhang, X. D., Chen, Z., and Ma, W. Y. (2005), *A study of relevance propagation for web search*. Proceedings of SIGIR 2005.

Baeza-Yates R. and Ribeiro-Neto B. (1999) *Modern Information Retrieval*. ACM Press, New York.

Salton, G., Wong, A., and Yang, C.S. (1975), *A Vector Space Model for Automatic Indexing*, Communications of the ACM, vol. 18, nr. 11, pages 613–620.

Morissette, M.-A. (2007), *CanLII's search engine*, Proceedings of the 8th conference Law via the Internet (Montreal, Canada)

Brin, S. and Page L. (1998), *The anatomy of a large-scale hypertextual Web search engine*, In Proceedings of the 7th International World Wide Web Conference (Brisbane, Australia, Apr. 14 –18). pp. 107–117.